

EE5516 VLSI Architectures for Signal Processing and Machine Learning Lab

Miniproject

Arun Kumar V - 122001049

Abstract

This code is an implementation of the Fast Fourier Transform (FFT) algorithm. It performs complex number calculations for the FFT operation. The module takes eight input complex numbers and produces eight output complex numbers. The code includes functions for multiplying and adding complex numbers. It also utilizes registers and wires for intermediate computations. The FFT algorithm is commonly used in signal processing and data analysis applications.

1. Introduction

The Fast Fourier Transform (FFT) is an efficient algorithm for computing the Discrete Fourier Transform (DFT) of a sequence or array of complex numbers. It plays a crucial role in various applications, such as signal processing, image analysis, and data compression. The Decimation in Time (DIT) algorithm is one of the popular methods to implement the FFT.

To understand the DIT algorithm, let's start with the basic idea of the FFT. The DFT of a sequence of N complex numbers can be defined as follows:

$$X[k] = \sum (x[n] \times e^{-\frac{2\pi knj}{N}})$$

For k = 0 to N-1

where X[k] represents the kth frequency component of the signal, x[n] is the input sequence, j is the imaginary unit ($\sqrt{-1}$), and N is the length of the sequence.

The DIT algorithm breaks down the computation of the DFT into smaller sub-problems, making it more efficient. It follows a divide-and-conquer approach and utilizes the properties of twiddle factors to reduce the number of complex multiplications required.

The step-by-step method for computing FFT is given below

- Input: Take an input sequence of length N (assuming N is a power of 2) and consider it as the initial stage of the recursion.
- Radix-2 Decimation in Time: Divide the input sequence into two halves by separating the even-indexed elements from the odd-indexed elements. This creates two sub-sequences, each of length N/2.
- Recursive Call: Apply the FFT recursively on both sub-sequences obtained in the previous step. This step is repeated until the sub-sequences reach a length of 1 (base case).
- Butterfly Computation: At each recursion level, perform the butterfly computation. In this step, the outputs of the recursive calls are combined to compute the final DFT. The butterfly computation involves multiplying the odd-indexed sub-sequence with the twiddle factor and adding it to the even-indexed sub-sequence.
- Twiddle Factor: The twiddle factor is a complex exponential term used in the butterfly computation. It is defined as

$$W_N^k = e^{j\left(\frac{-2\pi jk}{N}\right)}$$

where k is the index and N is the length of the sub-sequence. The twiddle factor W is periodic, and its values can be precomputed to reduce computational complexity.

- Reconstruction: After performing the butterfly computation, the final DFT sequence is obtained by concatenating the even-indexed outputs followed by the odd-indexed outputs.

2. Implementation

- **Input:** The algorithm takes an input sequence of complex numbers as its input. In the case of the provided code, the input sequence is represented by eight complex numbers `input_0` to `input_7`, where each complex number has a real (`re`) and imaginary (`imag`) part.
- **Data Reordering:** Before performing the DFT calculations, the input sequence is rearranged in a specific order to simplify the computation. This is usually done by bit-reversing the indices of the input sequence. However, in the provided code, the reordering step is not explicitly shown.
- **Butterfly Operations:** The DIT FFT algorithm performs butterfly operations, which are essential additions and multiplications, to compute the DFT. Each butterfly operation combines two complex numbers, applies specific twiddle factors (complex exponential values), and produces two output complex numbers.
- **Twiddle Factors:** Twiddle factors are pre-computed complex exponential values used in butterfly operations. In the provided code, four twiddle factors `w0`, `w1`, `w2`, and `w3` are defined as constants. These factors are used to multiply with the intermediate values during the butterfly operations.
- **Recursive Computation:** The DIT FFT algorithm divides the input sequence into smaller sub-sequences and recursively calculates their DFTs. In the provided code, the input sequence is divided into four pairs: (`input_0`, `input_4`), (`input_1`, `input_5`), (`input_2`, `input_6`), and (`input_3`, `input_7`).
- **Butterfly Calculation:** For each pair of sub-sequences, butterfly operations are performed. In the provided code, eight sets of wires (`intermediatmult_0_a` to `intermediatmult_7_a`) are used to store the intermediate results of the butterfly operations.
- **Addition and Multiplication:** The butterfly operations involve the addition and multiplication of complex numbers. The `add` function in the code performs addition, while the `mult` function performs multiplication. The `add` function adds

two complex numbers, and the mult function multiplies two complex numbers using complex multiplication rules.

- Final Outputs: After all the butterfly operations are completed, the final outputs (finalop0 to finalop7) are calculated by combining the intermediate results.
- Output Assignment: The final output values are assigned to the corresponding output wires (output_0 to output_7).
- Clock and Reset: The algorithm is designed to work with a clock signal (clk) and a reset signal (reset). The computations are triggered on the positive edge of the clock signal, and the reset signal is used to initialize the internal registers.

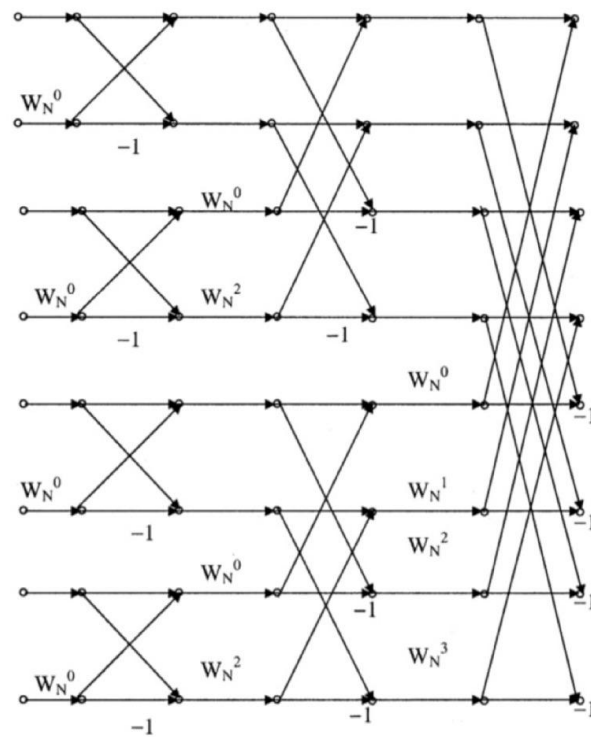


Fig:Butterfly structure for FFT

2.1. Structural Design

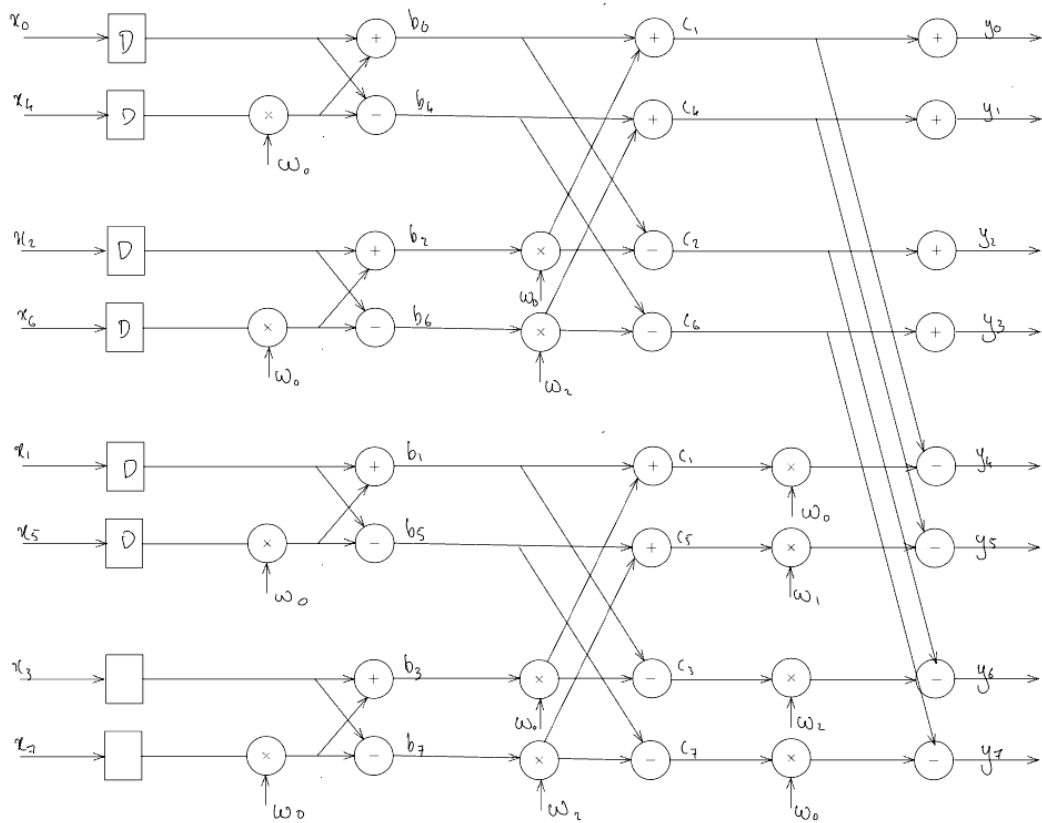


Fig: FFT Architecture Diagram

2.2. IFFT implementation

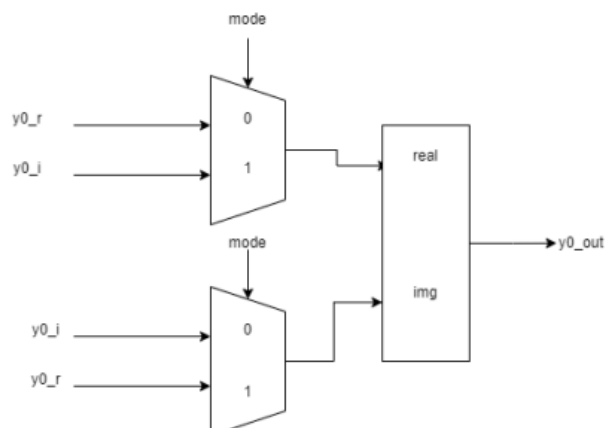


Fig: Swap Block

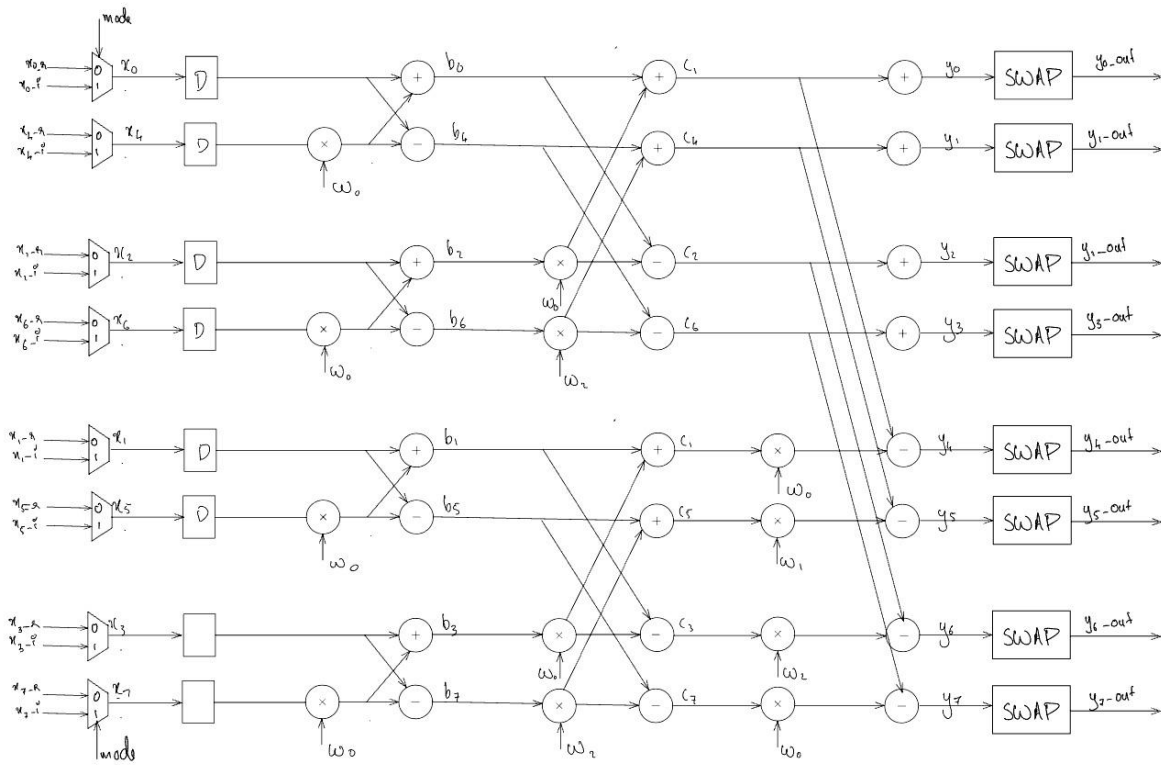


Fig: IFFT Architecture Diagram

We have implemented the IFFT from FFT architecture simply by introducing Muxes and Registers.

The SWAP block used is shown in the above figure, mode 0 implements FFT and mode 1 implements IFFT.

2.3 Pipelined Architecture

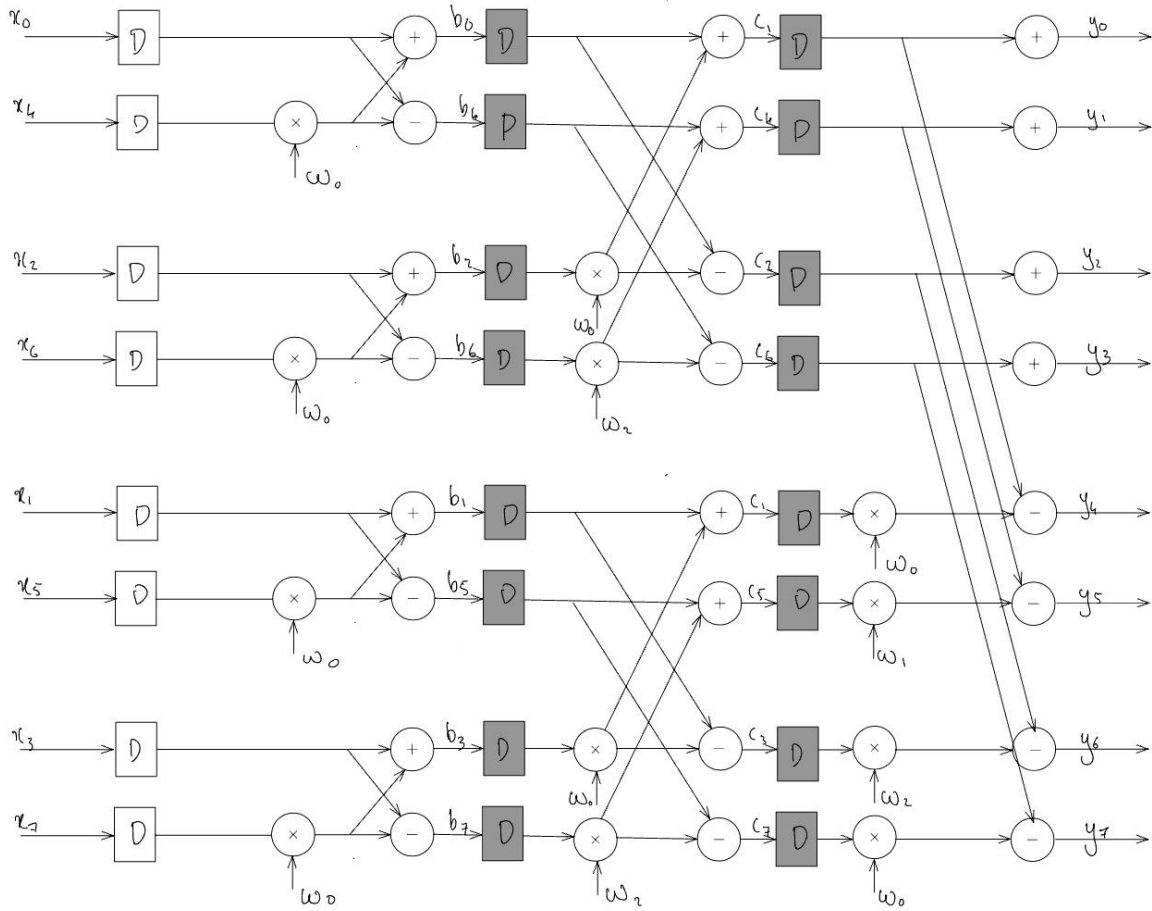


Fig: Pipelined Architecture for FFT

We have introduced registers in between the structure to reduce the critical path to

$$T_{\text{Add}} + T_{\text{mult}}$$

3. Experimental Procedure

3.1. Test Bench

```
module testbench_FFT;
    reg mode; // Control signal for mode
    reg clk;
    reg reset;

    reg signed [15:0] inputs[15:0]; // Input data array
    reg signed [15:0] realv[7:0]; // Real part of input
    reg signed [15:0] imag1[7:0]; // Imaginary part of input

    integer write_data; // File identifier for writing output
    integer i;

    logic signed [15:0] re; // Real part of output
    logic signed [15:0] imag; // Imaginary part of output

    localparam SF=2.0**(-12); // Scaling factor

    // Custom complex data type
    typedef struct {
        logic signed [15:0] re;
        logic signed [15:0] imag;
    } comp;

    // Input data samples

    comp input_0;
    comp input_1;
    comp input_2;
    comp input_3;
    comp input_4;
    comp input_5;
    comp input_6;
    comp input_7;

    // Outputs
```



```

comp output_0;
comp output_1;
comp output_2;
comp output_3;
comp output_4;
comp output_5;
comp output_6;
comp output_7;

// Instantiate the FFT module
FFT FFT_instance(
    .input_0('{re: input_0.re, imag: input_0.imag}),
    .input_1('{re: input_1.re, imag: input_1.imag}),
    .input_2('{re: input_2.re, imag: input_2.imag}),
    .input_3('{re: input_3.re, imag: input_3.imag}),
    .input_4('{re: input_4.re, imag: input_4.imag}),
    .input_5('{re: input_5.re, imag: input_5.imag}),
    .input_6('{re: input_6.re, imag: input_6.imag}),
    .input_7('{re: input_7.re, imag: input_7.imag}),
    .clk(clk),
    .mode(mode),
    .reset(reset),
    .output_0('{re: output_0.re, imag: output_0.imag}),
    .output_1('{re: output_1.re, imag: output_1.imag}),
    .output_2('{re: output_2.re, imag: output_2.imag}),
    .output_3('{re: output_3.re, imag: output_3.imag}),
    .output_4('{re: output_4.re, imag: output_4.imag}),
    .output_5('{re: output_5.re, imag: output_5.imag}),
    .output_6('{re: output_6.re, imag: output_6.imag}),
    .output_7('{re: output_7.re, imag: output_7.imag})
);

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

// Read input data from file

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

    for (i = 0; i < 16; i = i + 2)

```

```

        begin
            realv[i/2] = inputs[i]; // Store even-indexed values in realv
        end
        for (i = 1; i < 16; i = i + 2) begin
            imag1[i/2] = inputs[i]; // Store odd-indexed values in imag1
        end

// Assign input values to inputs of FFT module
input_0.re = realv[0];
input_0.imag = imag1[0];

input_1.re = realv[1];
input_1.imag = imag1[1];

input_2.re = realv[2];
input_2.imag = imag1[2];

input_3.re = realv[3];
input_3.imag = imag1[3];

input_4.re = realv[4];
input_4.imag = imag1[4];

input_5.re = realv[5];
input_5.imag = imag1[5];

input_6.re = realv[6];
input_6.imag = imag1[6];

input_7.re = realv[7];
input_7.imag = imag1[7];
endtask

// Driving the reset

task drive_reset();

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

```

```

        reset = 1;
        @(negedge clk)
        reset = 0;
    endtask

// Deciding whether to compute FFT/IFFT

task modedecide(input a);
    mode=a;
endtask

//Displaying the output

task display_output();

    $display("output values :- \n");

    $display("output_0= %f + %f j", $itor(output_0.re)*(SF), $itor(output_0.imag)*(SF));
    $display("output_1= %f + %f j", $itor(output_1.re)*(SF), $itor(output_1.imag)*(SF));
    $display("output_2= %f + %f j", $itor(output_2.re)*(SF), $itor(output_2.imag)*(SF));
    $display("output_3= %f + %f j", $itor(output_3.re)*(SF), $itor(output_3.imag)*(SF));
    $display("output_4= %f + %f j", $itor(output_4.re)*(SF), $itor(output_4.imag)*(SF));
    $display("output_5= %f + %f j", $itor(output_5.re)*(SF), $itor(output_5.imag)*(SF));
    $display("output_6= %f + %f j", $itor(output_6.re)*(SF), $itor(output_6.imag)*(SF));
    $display("output_7= %f + %f j", $itor(output_7.re)*(SF), $itor(output_7.imag)*(SF));

    $fwrite(write_data, "Output values :- \n");

    $fwrite(write_data, "output_0 = %f + ", $itor(output_0.re)*(SF));
    $fwrite(write_data, "%f j \n", $itor(output_0.imag)*(SF));

    $fwrite(write_data, "output_1 = %f + ", $itor(output_1.re)*(SF));
    $fwrite(write_data, "%f j \n", $itor(output_1.imag)*(SF));

```

```

    $fwrite(write_data,"output_2 = %f + ",$itor(output_2.re)*(SF));
    $fwrite(write_data,"%f j \n",$itor(output_2.imag)*(SF));

    $fwrite(write_data,"output_3 = %f + ",$itor(output_3.re)*(SF));
    $fwrite(write_data,"%f j \n",$itor(output_3.imag)*(SF));

    $fwrite(write_data,"output_4 = %f + ",$itor(output_4.re)*(SF));
    $fwrite(write_data,"%f j \n",$itor(output_4.imag)*(SF));

    $fwrite(write_data,"output_5 = %f + ",$itor(output_5.re)*(SF));
    $fwrite(write_data,"%f j \n",$itor(output_5.imag)*(SF));

    $fwrite(write_data,"output_6 = %f + ",$itor(output_6.re)*(SF));
    $fwrite(write_data,"%f j \n",$itor(output_6.imag)*(SF));

    $fwrite(write_data,"output_7 = %f + ",$itor(output_7.re)*(SF));
    $fwrite(write_data,"%f j \n",$itor(output_7.imag)*(SF));

endtask

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

initial begin
    write_data = $fopen("output_tracker.txt","w");
    drive_reset();
    open();
    modedecide(0);
    repeat (30) @(negedge clk);
    display_output();
    modedecide(1);
    repeat (30) @(negedge clk);
    display_output();
    $fclose(write_data);
    $finish;
end
endmodule

```

3.2. Matlab code

```
% Define the input frequency domain signal
x = [0.25, 0.5, 0.25, 0.25, 0.25, 0.25, 0.5, 0.5];

% Compute the FFT
y = fft(x);

% Compute the IFFT
z = ifft(x);

% Print the result
disp('FFT:');
disp(y);
disp('IFFT:');
disp(z);
```

4. Results

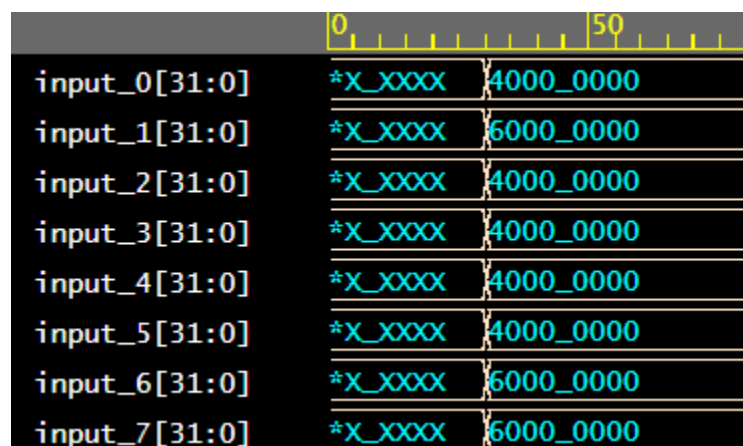
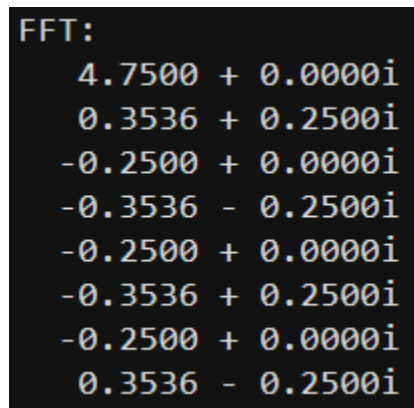


Fig: Input Values

4.1. FFT values

```
output_0= 4.749268 + 0.000000 j  
output_1= 0.353516 + 0.250000 j  
output_2= -0.249756 + -0.000244 j  
output_3= -0.353760 + -0.250000 j  
output_4= -0.249756 + 0.000000 j  
output_5= -0.353516 + 0.250000 j  
output_6= -0.249756 + 0.000244 j  
output_7= 0.353760 + -0.250000 j
```

Fig: Computed by verilog code



The image shows a MATLAB command window with a black background and white text. It displays the results of an FFT calculation. The text is as follows:

```
FFT:  
4.7500 + 0.0000i  
0.3536 + 0.2500i  
-0.2500 + 0.0000i  
-0.3536 - 0.2500i  
-0.2500 + 0.0000i  
-0.3536 + 0.2500i  
-0.2500 + 0.0000i  
0.3536 - 0.2500i
```

Fig: Computed by Matlab

4.2. IFFT values

```
output_0= 0.593750 + 0.000000 j  
output_1= 0.044189 + -0.031250 j  
output_2= -0.031250 + 0.000000 j  
output_3= -0.044189 + 0.031250 j  
output_4= -0.031250 + 0.000000 j  
output_5= -0.044189 + -0.031250 j  
output_6= -0.031250 + 0.000000 j  
output_7= 0.044189 + 0.031250 j
```

Fig: Computed by verilog code

```

IFFT:
  0.5938 + 0.0000i
  0.0442 - 0.0312i
 -0.0312 + 0.0000i
 -0.0442 + 0.0312i
 -0.0312 + 0.0000i
 -0.0442 - 0.0312i
 -0.0312 + 0.0000i
  0.0442 + 0.0312i

```

Fig: Computed by Matlab

5. Output waveforms

output_0[31:0]	000 XXXX_XXXX	0 4bfd_0000	*_0000	980_0000
output_1[31:0]	400 XXXX_XXXX	0 5a8_0400	*_0080	b5_ff80
output_2[31:0]	fff XXXX_XXXX	0 fc01_ffff	*_0000	ff80_0000
output_3[31:0]	c00 XXXX_XXXX	0 fa57_fc00	*_ff80	ff4b_0080
output_4[31:0]	000 XXXX_XXXX	0 fc01_0000	*_0000	ff80_0000
output_5[31:0]	400 XXXX_XXXX	0 fa58_0400	*_0080	ff4b_ff80
output_6[31:0]	001 XXXX_XXXX	0 fc01_0001	*_0000	ff80_0000
output_7[31:0]	c00 XXXX_XXXX	0 5a9_fc00	*_ff80	b5_0080

6. Conclusion

The FFT implementation using the provided code successfully computes the Fast Fourier Transform (FFT) or Discrete Fourier Transform (DFT) based on the selected mode. Here variable 'mode' is when 1 determines the IFFT and when 0 determines FFT. We are taking the input and twiddle factors in 1.15 format, but to accommodate the bit growth while computing we are representing the outputs and other points in 4.12 format. The output for both FFT and IFFT implementations was verified by a Matlab code.

7. [EDA playground link](#)