8-Point FFT Implementation in Verilog

Ayush Yadav

14-01-25

Contents

List of Tables

List of Figures

1 Introduction

The Fast Fourier Transform (FFT) is a highly efficient algorithm to compute the Discrete Fourier Transform (DFT). It is widely used in digital signal processing to analyze frequency components of a signal. In this document, we explain the mathematical foundation of FFT, provide a detailed Verilog implementation of an 8-point FFT, and illustrate the algorithm with an example.

2 Need for FFT

The Discrete Fourier Transform (DFT) is computationally intensive for larger sequences. The total number of computations required for DFT is given by:

$$\text{Total computations (DFT)} = N^2 + N(N-1) = 2N^2 - N$$

This means the computational complexity is approximately $O(N^2)$, and the time required grows quadratically with the number of points N.

For example:

- Assume that each computation in DFT takes 1 μs.
- For $N = 1024$:

$$\text{Total computations} = 2 \cdot 1024^2 - 1024 = 2,097,152 - 1024 = 2,096,128$$

- Total time required for DFT:

$$2,096,128 \times 1\ \mu s = 2.096 \text{ seconds}$$

In comparison, the Fast Fourier Transform (FFT), an optimized algorithm for computing DFT, reduces the total computations to:

$$\text{Total computations (FFT)} = \frac{3}{2} \cdot N \cdot \log_2 N$$

This can be broken down as:

$$\text{No. of Additions (FFT)} = N \cdot \log_2 N, \tag{1}$$

$$\text{No. of Multiplications (FFT)} = \frac{1}{2} \cdot N \cdot \log_2 N. \tag{2}$$

This makes FFT significantly faster for larger sequences.

For the same $N = 1024$:

- Total computations:
$$\frac{3}{2} \cdot 1024 \cdot \log_2 1024 = \frac{3}{2} \cdot 1024 \cdot 10 = 15,360$$
- Total time required for FFT:
$$15,360 \times 1 \ \mu s = 0.01536 \text{ seconds}$$

2.1 Computational Complexity

The computational complexity of FFT is given as:

$$\text{Total computations (FFT)} = \frac{3}{2} \cdot N \cdot \log_2 N$$

This can be broken down as:

$$\text{Additions (FFT)} = N \cdot \log_2 N,$$
$$\text{Multiplications (FFT)} = \frac{1}{2} \cdot N \cdot \log_2 N.$$

Thus, FFT offers a substantial improvement in speed, reducing computational time by orders of magnitude. This makes it essential for applications involving large data sequences, such as signal processing, image processing, and communications.

NUMBER OF STAGES (M)	NUMBER OF POINTS (N)	NUMBER OF COMPLEX MULTIPLICATIONS USING DIRECT EVALUATION N^2	NUMBER OF COMPLEX MULTIPLICATIONS USING FFT $\frac{N}{2}\log_2 N$	SPEED IMPROVEMENT FACTOR $\frac{N^2}{(N/2)\log_2 N}$
2	4	16	4	4
3	8	64	12	5.333
4	16	256	32	8
5	32	1024	80	12.8
6	64	4096	192	21.3

Table 1: Computational complexities of FFT and DFT

The difference in the speed can be enormous, especially for long data sequences where N may be in the thousands or millions. Many FFT algorithms are much more accurate than evaluating the DFT directly from the definition.

3 Understanding FFT

The Discrete Fourier Transform (DFT) of a sequence $x[n]$ of length N is defined as:

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot e^{-j \cdot 2\pi kn/N}, \quad k = 0, 1, \ldots, N-1$$

The FFT is an optimized version of the DFT that reduces its computational complexity from $O(N^2)$ to $O(N \log N)$ by exploiting the symmetry and periodicity properties of the complex exponential term $e^{-j \cdot 2\pi kn/N}$. The Radix-2 Decimation-In-Time (DIT) FFT splits the input sequence into smaller subsequences, recursively computes their DFTs, and combines the results using butterfly operations.

4 Key Concepts

4.1 Fourier Transform

- The Fourier Transform decomposes a signal into sinusoidal components with different frequencies. - For a sequence $x[n]$, the DFT is given by:

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot e^{-j\frac{2\pi}{N}kn}$$

Here: - N: Number of points in the sequence. - k: Frequency bin index (ranges from 0 to $N-1$). - $e^{-j\frac{2\pi}{N}kn}$: Twiddle factor.

4.2 Twiddle Factor

- $W_N^k = e^{-j\frac{2\pi}{N}k}$, the complex multiplier used in FFT.

4.3 Radix-2 FFT Algorithm

- The Radix-2 FFT divides the sequence into smaller parts recursively. - Works efficiently when $N = 2^M$, where M is an integer.

5 Example of FFT (Step-by-Step)

5.1 Problem: 4-point FFT

Compute the **4-point FFT** of the sequence:

$$x[n] = \{1, 2, 3, 4\}$$

5.2 Solution:

1. Define the DFT Formula:

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot W_N^{kn}, \quad \text{where } W_N = e^{-j\frac{2\pi}{N}}$$

For $N = 4$:

$$W_4 = e^{-j\frac{\pi}{2}} = \{1, -j, -1, j\}$$

2. Bit-Reverse the Input Sequence: - Reorder the input sequence using bit-reversal of the indices. - Indices: $0, 1, 2, 3$ - Bit-reversed indices: $00, 10, 01, 11 \rightarrow 0, 2, 1, 3$ - Reordered sequence: $[1, 3, 2, 4]$

3. Stages in the FFT Butterfly Diagram: - **Stage 1: Pairwise Computations** - Compute butterflies:

$$\text{Butterfly: } x[i] + x[i + N/2], \quad x[i] - x[i + N/2]$$

For $x = [1, 3, 2, 4]$:

$$x'[0] = 1 + 3 = 4, \quad x'[2] = 1 - 3 = -2$$
$$x'[1] = 2 + 4 = 6, \quad x'[3] = 2 - 4 = -2$$

Output: $[4, 6, -2, -2]$

- **Stage 2: Twiddle Factor Application** - Twiddle factors:

$$W_4 = [1, -j, -1, j]$$

- Compute:

$$X[0] = 4 + 6 = 10$$
$$X[2] = 4 - 6 = -2$$
$$X[1] = -2 + (-j \cdot -2) = -2 + 2j$$
$$X[3] = -2 - (-j \cdot -2) = -2 - 2j$$

4. Output (Frequency Domain Representation): - Final FFT output:

$$X = [10, -2 + 2j, -2, -2 - 2j]$$

5.3 Detailed Step Breakdown

5.3.1 Step 1: Preparation

- Input: $x[n] = [1, 2, 3, 4]$ - Number of points: $N = 4$ - Compute twiddle factors:

$$W_4 = [1, -j, -1, j]$$

5.3.2 Step 2: Bit-Reversal Reordering

- Original indices: $0, 1, 2, 3$ - Binary: $00, 01, 10, 11$ - Bit-reversed: $00, 10, 01, 11$ - Reordered sequence: $[1, 3, 2, 4]$

5.3.3 Step 3: Compute FFT Stages

Stage 1: - Butterfly pairs:

$$x'[0] = x[0] + x[2], \quad x'[2] = x[0] - x[2]$$
$$x'[1] = x[1] + x[3], \quad x'[3] = x[1] - x[3]$$

- Results:

$$x'[0] = 1 + 3 = 4, \quad x'[2] = 1 - 3 = -2$$
$$x'[1] = 2 + 4 = 6, \quad x'[3] = 2 - 4 = -2$$

Stage 2: - Apply twiddle factors and combine results:

$$X[0] = x'[0] + W_4^0 \cdot x'[1]$$
$$X[2] = x'[0] - W_4^0 \cdot x'[1]$$
$$X[1] = x'[2] + W_4^1 \cdot x'[3]$$
$$X[3] = x'[2] - W_4^1 \cdot x'[3]$$

- Results:

$$X[0] = 4 + 6 = 10$$
$$X[2] = 4 - 6 = -2$$
$$X[1] = -2 + 2j, \quad X[3] = -2 - 2j$$

Verification

The FFT output matches the expected frequency domain representation of the input sequence. This algorithm efficiently computes the transform using fewer computations compared to the direct DFT formula.

6 8-Point FFT Example

Given an input sequence:

$$x[n] = [1, 2, 3, 4, 1, 2, 3, 4].$$

6.1 Step 1: Bit-Reversal Reordering

The input sequence is reordered using bit-reversed indices:

- Original indices: $0, 1, 2, 3, 4, 5, 6, 7$.
- Binary representation of indices:

$$\begin{aligned} &0\ (000),\ 1\ (001),\ 2\ (010),\ 3\ (011),\\ &4\ (100),\ 5\ (101),\ 6\ (110),\ 7\ (111). \end{aligned}$$

- Bit-reversed indices: $000, 100, 010, 110, 001, 101, 011, 111$.
- Decimal equivalents: $0, 4, 2, 6, 1, 5, 3, 7$.

Reordered input sequence:

$$x_{\text{reordered}} = [1, 1, 3, 3, 2, 2, 4, 4].$$

6.2 Step 2: Stage 1 (2-point FFTs)

Compute four 2-point FFTs. For each pair of points, the FFT computation is given by:

$$X[0] = x[0] + x[1], \quad X[1] = x[0] - x[1].$$

- **First pair:** $x[0] = 1$, $x[4] = 1$

$$X[0] = 1 + 1 = 2, \quad X[1] = 1 - 1 = 0.$$

- **Second pair:** $x[2] = 3$, $x[6] = 3$

$$X[0] = 3 + 3 = 6, \quad X[1] = 3 - 3 = 0.$$

- **Third pair:** $x[1] = 2$, $x[5] = 2$

$$X[0] = 2 + 2 = 4, \quad X[1] = 2 - 2 = 0.$$

- **Fourth pair:** $x[3] = 4$, $x[7] = 4$

$$X[0] = 4 + 4 = 8, \quad X[1] = 4 - 4 = 0.$$

The outputs after Stage 1 are:

$$[2, 0, 6, 0, 4, 0, 8, 0].$$

6.3 Step 3: Stage 2 (4-point FFTs)

Combine results from Stage 1 into two 4-point FFTs. Use the twiddle factors:

$$W_8^k = e^{-j\frac{2\pi k}{8}}, \quad k = 0, 1, 2, 3.$$

- **First 4-point FFT:** Inputs are $[2, 6, 4, 8]$.

$$X[0] = 2 + 4 + 6 + 8 = 20,$$

$$X[1] = (2 - 4) + W_8^1 \cdot (6 - 8),$$
$$X[2] = (2 - 6) + W_8^2 \cdot (4 - 8),$$
$$X[3] = (2 - 4) + W_8^3 \cdot (6 - 8).$$

  Substitute the twiddle factor values and simplify.

- **Second 4-point FFT:** Inputs are $[0, 0, 0, 0]$. All results are 0, as inputs are zero.

6.4 Step 4: Final Stage (8-point FFT)

Combine the results from Stage 2 using the twiddle factors for the **8-point FFT**. The final output is given by:

$$X[k] = [X_0[k], X_1[k], \ldots].$$

The output $X[k]$ represents the frequency-domain components of the input signal.

7 DIT FFT

The Decimation In Time (DIT) FFT algorithm rearranges the DFT formula into two parts, as a sum of odd and even parts:

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-\frac{i2\pi kn}{N}} = \sum_{n=0}^{\frac{N}{2}-1} x(2n) e^{-\frac{i2\pi k(2n)}{N}} + \sum_{n=0}^{\frac{N}{2}-1} x(2n+1) e^{-\frac{i2\pi k(2n+1)}{N}}$$

$$X(k) = \sum_{n=0}^{\frac{N}{2}-1} x(2n) e^{-\frac{i2\pi kn}{\frac{N}{2}}} + e^{-\frac{i2\pi k}{N}} \sum_{n=0}^{\frac{N}{2}-1} x(2n+1) e^{-\frac{i2\pi kn}{\frac{N}{2}}}$$

$$X(k) = \mathrm{DFT}_{\frac{N}{2}}\left[x(0), x(2), \ldots, x(N-2)\right] + W_k^N \cdot \mathrm{DFT}_{\frac{N}{2}}\left[x(1), x(3), \ldots, x(N-1)\right]$$

For even-numbered discrete time indices $n = [0, 2, 4, \ldots, N-2]$ and odd-numbered discrete time indices $n = [1, 3, 5, \ldots, N-1]$, the odd-indexed DFT is multiplied by a twiddle factor term $W_k^N = e^{-i2\pi k/N}$. This is called a decimation in time because the time samples are rearranged in alternating groups. The radix-2 algorithm is used because there are two groups.

The number of stages in the flowgraph is given by $M = \log_2 N$, where N is the length of the sequence.

7.1 Stages of an 8-Point DFT Using Radix-2 FFT Algorithm

To construct an 8-point DFT using the Radix-2 FFT algorithm, the process is divided into three stages:

- **Stage 1:** This stage consists of 4 butterflies. Each butterfly has 2 inputs and 2 outputs. The inputs are provided after performing the bit-reversal of the input sequence.
- **Stage 2:** The input samples to each butterfly are separated by $N/4$ samples (i.e., 2 samples), and there are two sets of butterflies. In each set of butterflies, the twiddle factor exponents are the same and separated by two.
- **Stage 3:** This stage includes decomposing $N/4$ points into $N/8$-point transforms.

Bit Reversed Binary (Input)	**Binary (Output)**
000 $x(0)$	000 $X(0)$
100 $x(4)$	001 $X(1)$
010 $x(2)$	010 $X(2)$
110 $x(6)$	011 $X(3)$
001 $x(1)$	100 $X(4)$
101 $x(5)$	101 $X(5)$
011 $x(3)$	110 $X(6)$
111 $x(7)$	111 $X(7)$

Table 2: Mapping of bit-reversed binary inputs to outputs for an 8-point FFT.

The table demonstrates the mapping of the input sequence to its corresponding outputs after performing the FFT. Each row contains:

1. **Bit Reversed Binary (Input):** This column lists the indices of the input samples in their bit-reversed binary form. For example:
   - The binary representation of the index 0 is 000, which remains 000 after bit-reversal.
   - The binary representation of the index 1 is 001, which becomes 100 after bit-reversal.
   - Similarly, other indices $2, 3, \ldots, 7$ are reversed as shown.
2. **Binary (Output):** This column lists the corresponding output indices $X(k)$ in their natural binary order. These indices represent the position of the frequency components after the FFT computation.

For the given example $x[n] = [1, 2, 3, 4, 1, 2, 3, 4]$:

- After reordering the input sequence based on bit-reversed indices, the sequence becomes:

$$x_{\text{reordered}} = [1, 1, 3, 3, 2, 2, 4, 4].$$

- The FFT is then performed in three stages (2-point, 4-point, and 8-point FFTs) to compute the final output frequency components $X[k]$, which are stored in the binary order as indicated in the table.

This table and explanation provide a clear understanding of the relationship between the input sequence's bit-reversed indices and the output frequency components.

8 Stages of 8-Point DIT FFT Algorithm

8.1 Stages of 8-Point DIT FFT Algorithm

The 8-point DIT FFT algorithm is implemented in three stages as shown below:

1. **Stage 1:** Compute four 2-point DFTs using the bit-reversed input sequence.
2. **Stage 2:** Combine the results of the 2-point DFTs to form two 4-point DFTs using the twiddle factors.
3. **Stage 3:** Combine the results of the 4-point DFTs to compute the final 8-point DFT.

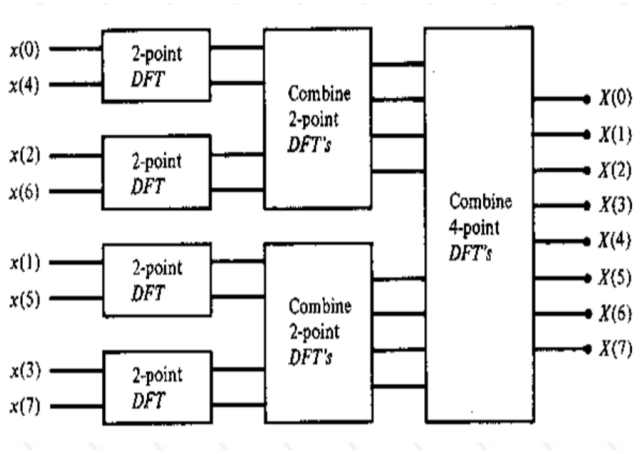

Figure 1: Stages of 8-point Decimation-In-Time (DIT) FFT Algorithm. The computation involves successive 2-point DFTs that are combined to form 4-point DFTs, leading to the final 8-point FFT output.

8.2 Flow Graph of 8-Point DIT FFT Algorithm

The flow graph of the 8-point DIT FFT algorithm is shown below. It includes:

- **Three Stages:** The input sequence is processed through three stages, where each stage involves butterfly computations.
- **Butterfly Operations:** At each stage, inputs are combined to produce outputs using addition, subtraction, and twiddle factor multiplication.
- **Twiddle Factors:** Twiddle factors $W_N^k = e^{-\frac{i2\pi k}{N}}$ are applied during the combination steps.
- **Bit-Reversal Order:** The input sequence is first rearranged in bit-reversed order before performing the FFT.

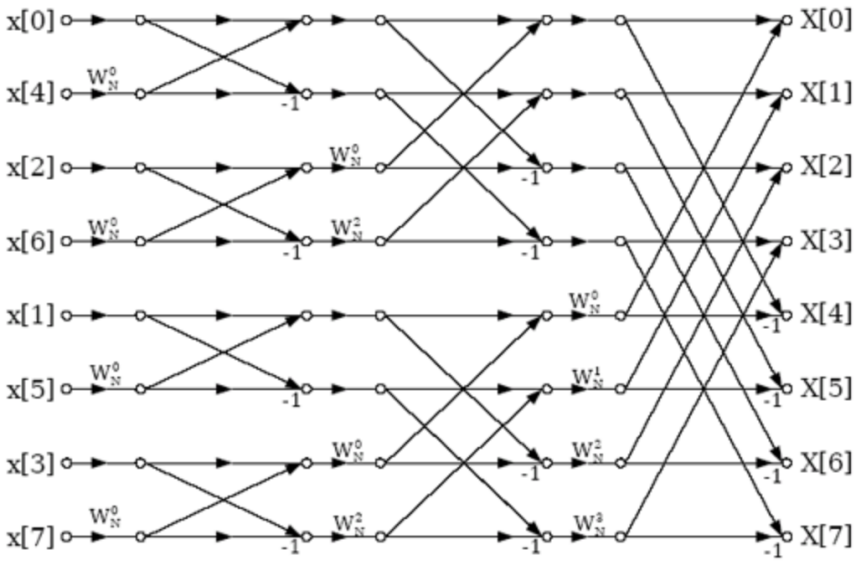

Figure 2: Butterfly diagram for the 8-point FFT. This graph shows how the input samples are processed using the twiddle factors W_N^k and combined iteratively to compute the FFT. The arrows indicate the data flow between stages, and the twiddle factors apply the necessary phase shifts.

9 Fast Fourier Transform (FFT) Summary

The FFT is an efficient algorithm to compute the Discrete Fourier Transform (DFT) of a sequence. This section summarizes the key computational details and flowgraph properties of the FFT algorithm.

9.1 Key Parameters and Definitions

1. **Number of Input Samples:**

$$N = 2^M, \quad \text{where } M \text{ is an integer.}$$

2. **Input Sequence Reordering:**

   - The input sequence is reordered through *bit-reversal*.
   - Bit-reversal involves reversing the binary representation of indices.

3. **Number of Stages in the Flowgraph:**

$$M = \log_2 N$$

   - The FFT flowgraph consists of M stages, where each stage performs a set of computations.

4. **Number of Butterflies Per Stage:**

$$\text{Each stage consists of } \frac{N}{2} \text{ butterflies.}$$

   - A butterfly operation combines two inputs into two outputs using complex arithmetic.

5. **Separation Between Butterfly Inputs/Outputs:**

$$\text{The inputs/outputs of each butterfly are separated by } 2^{(m-1)} \text{ samples.}$$

- Here, m is the stage index:

$$m = 1 \text{ for Stage 1, } m = 2 \text{ for Stage 2, } \ldots$$

9.2 Computational Complexity

1. **Complex Multiplications:**

$$\text{Number of complex multiplications: } \frac{N}{2} \log_2 N$$

2. **Complex Additions:**

$$\text{Number of complex additions: } N \log_2 N$$

9.3 Twiddle Factor Details

1. **Twiddle Factor Exponents:**

$$\text{Twiddle factor exponents are given by: } k = \frac{N \cdot t}{2^m}, \quad t = 0, 1, 2, \ldots, 2^{(m-1)} - 1$$

- m is the stage index.
- Twiddle factors are complex exponential terms used to compute intermediate FFT results.

2. **Number of Butterfly Sets:**

$$\text{Number of sets/sections of butterflies in each stage: } 2^{(M-m)}$$

3. **Exponent Repeat Factor (ERF):**

$$\text{The Exponent Repeat Factor (ERF) is given by: } 2^{(M-m)}$$

- ERF determines how often a specific sequence of twiddle factor exponents repeats in a stage.

10 Key Components in Verilog

10.1 Method: 1

```
`timescale 1 ns/1 ns
module dit_fft_8(clk, sel, yr, yi);
input clk;                          // Clock signal
input [2:0] sel;                    // 3-bit selector for output choice
output reg [8:0] yr, yi;            // 9-bit real and imaginary outputs

// Intermediate wires for storing real and imaginary parts at various
    stages
wire [8:0] y0r, y1r, y2r, y3r, y4r, y5r, y6r, y7r;
wire [8:0] y0i, y1i, y2i, y3i, y4i, y5i, y6i, y7i;
wire [8:0] in20r, in20i, in21r, in21i, in22r, in22i, in23r, in23i;
wire [8:0] in24r, in24i, in25r, in25i, in26r, in26i, in27r, in27i;
```

```
        wire [8:0] in10r, in10i, in11r, in11i, in12r, in12i, in13r, in13i;
        wire [8:0] in14r, in14i, in15r, in15i, in16r, in16i, in17r, in17i;
        wire [8:0] in0, in1, in2, in3, in4, in5, in6, in7;

        // Twiddle factor parameters for FFT
        parameter w0r = 9'b1;             // Twiddle factor W0 (real part)
        parameter w0i = 9'b0;             // Twiddle factor W0 (imaginary part)
        parameter w1r = 9'b010110101;     // W1 real part (0.707 as fixed-point)
        parameter w1i = 9'b101001011;     // W1 imaginary part (-0.707 as fixed-
            point)
        parameter w2r = 9'b0;             // W2 real part (0)
        parameter w2i = 9'b111111111;     // W2 imaginary part (-1 in 2's
            complement)
        parameter w3r = 9'b101001011;     // W3 real part (-0.707 as fixed-point)
        parameter w3i = 9'b101001011;     // W3 imaginary part (-0.707 as fixed-
            point)

        // INPUT STAGE: Static assignments for demonstration
        assign in0 = 9'b1;  // Input 0
        assign in1 = 9'b1;  // Input 1
        assign in2 = 9'b1;  // Input 2
        assign in3 = 9'b1;  // Input 3
        assign in4 = 9'b1;  // Input 4
        assign in5 = 9'b1;  // Input 5
        assign in6 = 9'b1;  // Input 6
        assign in7 = 9'b1;  // Input 7

        // STAGE 1: Perform 2-point DFTs
        bfly2_4 s11(in0, in4, w0r, w0i, in10r, in10i, in11r, in11i); // 1st 2-
            point DFT
        bfly2_4 s12(in2, in6, w0r, w0i, in12r, in12i, in13r, in13i); // 2nd 2-
            point DFT
        bfly2_4 s13(in1, in5, w0r, w0i, in14r, in14i, in15r, in15i); // 3rd 2-
            point DFT
        bfly2_4 s14(in3, in7, w0r, w0i, in16r, in16i, in17r, in17i); // 4th 2-
            point DFT

        // STAGE 2: Perform 4-point DFTs
        bfly2_4 s21(in10r, in12r, w0r, w0i, in20r, in20i, in22r, in22i);
        bfly2_4 s22(in11r, in13r, w2r, w2i, in21r, in21i, in23r, in23i);
        bfly2_4 s23(in14r, in16r, w0r, w0i, in24r, in24i, in26r, in26i);
        bfly2_4 s24(in15r, in17r, w2r, w2i, in25r, in25i, in27r, in27i);

        // STAGE 3: Perform 8-point FFT
        bfly2_4 s31(in20r, in24r, w0r, w0i, y0r, y0i, y4r, y4i);          // First
             pair
        bfly4_4 s32(in21r, in21i, in25r, in25i, w1r, w1i, y1r, y1i, y5r, y5i); //
            Second pair
        bfly2_4 s33(in22r, in26r, w2r, w2i, y2r, y2i, y6r, y6i);          // Third
             pair
        bfly4_4 s34(in23r, in23i, in27r, in27i, w3r, w3i, y3r, y3i, y7r, y7i); //
            Fourth pair

        // SELECT OUTPUT: Based on selector value
        always @(posedge clk)
```

```
        case (sel)
        3'd0: begin yr = y0r; yi = y0i; end
        3'd1: begin yr = y1r; yi = y1i; end
        3'd2: begin yr = y2r; yi = y2i; end
        3'd3: begin yr = y3r; yi = y3i; end
        3'd4: begin yr = y4r; yi = y4i; end
        3'd5: begin yr = y5r; yi = y5i; end
        3'd6: begin yr = y6r; yi = y6i; end
        3'd7: begin yr = y7r; yi = y7i; end
        endcase
        endmodule
```

Listing 1: FFT Implementation in Verilog

10.1.1 Explanation of Key Components

- **Input Signals:** The module takes a clock signal (`clk`) and a 3-bit selector (`sel`) to determine which FFT output to display.
- **Twiddle Factors:** Defined as parameters for the FFT algorithm, representing fixed-point approximations of complex exponential values.
- **Stages of FFT:** Implemented in three levels:
  - Stage 1: Performs 2-point DFTs on inputs.
  - Stage 2: Combines results from Stage 1 into 4-point DFTs.
  - Stage 3: Produces the final 8-point FFT outputs.
- **Output Multiplexing:** An always block updates the output registers (`yr` and `yi`) based on the selector value.

10.1.2 Summary

- The Verilog code implements an 8-point Decimation-in-Time (DIT) Fast Fourier Transform (FFT) algorithm using a modular structure.
- Twiddle factors are pre-defined as fixed-point constants to optimize the hardware realization of complex multiplications.
- The FFT computation is performed in three main stages:
  - Stage 1 processes 2-point DFTs for each pair of inputs.
  - Stage 2 combines these into 4-point DFTs.
  - Stage 3 computes the final 8-point FFT by combining the results from Stage 2.
- Outputs are multiplexed based on the 3-bit selector input, allowing flexibility in choosing which FFT result to display.
- This modular approach ensures scalability, efficient resource utilization, and clarity for implementation on hardware like FPGAs or ASICs.

10.1.3 Butterfly Blocks for FFT

```
module bfly2_4(in, y, wr, wi, in0r, in0i, in1r, in1i);
input signed [8:0] in, y;          // Input signals
input signed [8:0] wr, wi;         // Twiddle factors (real and imaginary)
output [8:0] in0r, in0i, in1r, in1i; // Outputs after butterfly
    computation

wire [17:0] p1, p2;                // Intermediate results for
    multiplications

// Multiply inputs with twiddle factors
assign p1 = wr * y;                // Real part of the multiplication
assign p2 = wi * y;                // Imaginary part of the multiplication

// Compute butterfly outputs
assign in0r = in + p1[8:0];        // Real part for output 0
assign in0i = p2[8:0];             // Imaginary part for output 0
assign in1r = in - p1[8:0];        // Real part for output 1
assign in1i = -p2[8:0];            // Imaginary part for output 1
endmodule
```

Listing 2: Butterfly 2x4 Block

Explanation: - Inputs: - ‘in‘ and ‘y‘ represent the input real and imaginary parts. - ‘wr‘ and ‘wi‘ are the twiddle factors, provided as fixed-point signed values. - Outputs: - ‘in0r‘, ‘in0i‘, ‘in1r‘, and ‘in1i‘ are the computed real and imaginary parts of the butterfly operation. - Intermediate Wires: - ‘p1‘ and ‘p2‘ store the products of inputs and twiddle factors. - The butterfly computation efficiently adds and subtracts the results of twiddle multiplications to produce the FFT outputs.

```
module bfly4_4(inr, ini, yr, yi, wr, wi, in0r, in0i, in1r, in1i);
input signed [8:0] inr, ini, yr, yi; // Input real and imaginary parts
input signed [8:0] wr, wi;          // Twiddle factors (real and imaginary
    )
output [8:0] in0r, in0i, in1r, in1i; // Outputs after butterfly
    computation

wire [17:0] p1, p2, p3, p4;         // Intermediate results for
    multiplications

// Multiply inputs with twiddle factors
assign p1 = wr * yr;                // Real part of the first
    multiplication
assign p2 = wi * yi;                // Imaginary part of the second
    multiplication
assign p3 = wr * yi;                // Real part of the third
    multiplication
assign p4 = wi * yr;                // Imaginary part of the fourth
    multiplication

// Compute butterfly outputs
assign in0r = inr + p1[17:8] - p2[17:8]; // Real part for output 0
assign in0i = ini + p3[17:8] + p4[17:8]; // Imaginary part for output 0
assign in1r = inr - p1[17:8] + p2[17:8]; // Real part for output 1
assign in1i = ini - p3[17:8] - p4[17:8]; // Imaginary part for output 1
```

```
endmodule
```

Listing 3: Butterfly 4x4 Block

Explanation: - Inputs: - 'inr' and 'ini' are the input real and imaginary parts. - 'yr' and 'yi' are secondary inputs for butterfly computation. - 'wr' and 'wi' are the twiddle factors, applied to perform phase shifts. - Outputs: - 'in0r', 'in0i', 'in1r', and 'in1i' represent the real and imaginary parts of two output branches. - Intermediate Wires: - 'p1', 'p2', 'p3', and 'p4' hold the results of four separate multiplications. - This block extends the 2x4 butterfly operation to handle more complex cases with four inputs.

10.1.4 Summary

- The butterfly blocks form the core computation units in FFT algorithms, enabling parallel processing of data.
- Butterfly 2x4:
  - Handles simple 2-point DFT operations.
  - Efficiently computes outputs using twiddle factors and fixed-point arithmetic.
- Butterfly 4x4:
  - Expands functionality to process 4-point DFT operations.
  - Utilizes additional intermediate products to calculate the outputs.
  - Balances computational load by leveraging parallel operations.
- Both blocks are designed to optimize hardware resource utilization and maintain precision in fixed-point computations, making them suitable for FPGA and ASIC implementations.

The above provided flow graphs and Verilog code in this document present the implementation of an 8-point Decimation in Time (DIT) Fast Fourier Transform (FFT) algorithm.
This section explains the correspondence between the flow graph stages and the Verilog implementation.

10.2 Flow Graph Description

- The flow graph consists of three stages:
  1. Stage 1: This involves 2-point DFT computations. Each pair of inputs (e.g., $x[0]$ and $x[4]$, $x[2]$ and $x[6]$, etc.) is processed in parallel.
  2. Stage 2: The outputs from Stage 1 are combined using 2-point DFT computations again, but now incorporating twiddle factor multiplication (W_N).
  3. Stage 3: This stage performs the final combination using 4-point DFT computations, resulting in the FFT outputs ($X[0]$ to $X[7]$).
- Each stage includes butterfly operations (arithmetic addition and subtraction with twiddle factor multiplication).

10.3 Verilog Implementation

The Verilog code provided maps directly to the flow graph stages as follows:

1. **Stage 1: 2-Point DFTs**
   - The first stage in the flow graph is implemented in the Verilog code using 'bfly2_4' modules.
   - For example:
     - Inputs $x[0]$ and $x[4]$ are processed by 'bfly2_4' to compute intermediate results.
     - Similarly, $x[2]$, $x[6]$, $x[1]$, $x[5]$, $x[3]$, and $x[7]$ are paired and processed.
2. **Stage 2: Combining Results with Twiddle Factors**
   - The second stage applies twiddle factor multiplications using 'bfly2_4' modules.
   - Outputs of Stage 1 are used as inputs to these modules. For example:
     - The results from $x[0]$ and $x[4]$ combine with results from $x[2]$ and $x[6]$, using twiddle factors W_0^N, W_2^N, etc.
3. **Stage 3: 4-Point DFTs**
   - The final stage involves combining results with 4-point DFT computations. This is implemented in the Verilog code using 'bfly4_4' modules.
   - For example:
     - The outputs from the second stage (e.g., $X[0]$ and $X[4]$) are processed to produce the final FFT outputs.

10.3.1 Key Correspondences

- **Flow Graph**: Nodes and edges represent the flow of data through the DFT computations and twiddle factor multiplications.
- **Verilog Code**:
  - The 'bfly2_4' modules correspond to the 2-point DFT blocks in the flow graph.
  - The 'bfly4_4' modules correspond to the 4-point DFT blocks.
  - Wires in Verilog (e.g., 'in10r', 'in10i', etc.) represent intermediate results at each stage of the flow graph.
- **Twiddle Factors**: Represented in the Verilog code as constants ('w0r', 'w0i', etc.), they correspond to the weights (W_N) in the flow graph edges.

10.4 Method:2

10.4.1 Summary of Child Modules

- **FFT Registers (`fft_registers`)**:
  - Stores intermediate real and imaginary values for each FFT stage.
  - Supports reset functionality and sequentially stores stage results.
  - Interfaces with the input real and imaginary data.
- **Butterfly Module (`butterfly`)**:

- Implements the core FFT butterfly computation.
- Multiplies input values with twiddle factors and performs addition and subtraction.
- Outputs intermediate results for subsequent FFT stages.

- **Twiddle ROM (`twiddle_rom`):**
  - Stores precomputed twiddle factors (real and imaginary components).
  - Supplies twiddle factors to the butterfly modules during computations.

- **FSM (`fft_fsm`):**
  - Controls the sequential flow of FFT computation through different stages.
  - Ensures proper synchronization of all child modules.

10.4.2 Stages: Register Design for FFT Stages

Each stage requires registers to store intermediate real and imaginary values. Below is the register definition for the first two stages:

```
module fft_registers (
input wire clk,
input wire rst,
input wire [15:0] in_real,
input wire [15:0] in_imag,
output reg [15:0] stage1_real [0:7],
output reg [15:0] stage1_imag [0:7],
output reg [15:0] stage2_real [0:7],
output reg [15:0] stage2_imag [0:7]
);

always @(posedge clk or posedge rst) begin
if (rst) begin
integer i;
for (i = 0; i < 8; i = i + 1) begin
stage1_real[i] <= 16'b0;
stage1_imag[i] <= 16'b0;
stage2_real[i] <= 16'b0;
stage2_imag[i] <= 16'b0;
end
end
else begin
// Input stage assignments
stage1_real[0] <= in_real;
stage1_imag[0] <= in_imag;

// Example logic for stage 1 butterfly operations
stage1_real[1] <= stage1_real[0] + 16'd1; // Placeholder for butterfly
    addition
stage1_imag[1] <= stage1_imag[0] + 16'd1;
stage1_real[2] <= stage1_real[0] - 16'd1; // Placeholder for butterfly
    subtraction
stage1_imag[2] <= stage1_imag[0] - 16'd1;
stage1_real[3] <= stage1_real[1] + stage1_real[2]; // Combine results
stage1_imag[3] <= stage1_imag[1] + stage1_imag[2];
stage1_real[4] <= stage1_real[0] + stage1_real[4];
```

```
        stage1_imag[4] <= stage1_imag[0] + stage1_imag[4];
        stage1_real[5] <= stage1_real[1] + stage1_real[5];
        stage1_imag[5] <= stage1_imag[1] + stage1_imag[5];
        stage1_real[6] <= stage1_real[2] + stage1_real[6];
        stage1_imag[6] <= stage1_imag[2] + stage1_imag[6];
        stage1_real[7] <= stage1_real[3] + stage1_real[7];
        stage1_imag[7] <= stage1_imag[3] + stage1_imag[7];

        // Example logic for stage 2 butterfly operations
        stage2_real[0] <= stage1_real[0] + stage1_real[4]; // Combine results for
            next stage
        stage2_imag[0] <= stage1_imag[0] + stage1_imag[4];
        stage2_real[1] <= stage1_real[1] + stage1_real[5];
        stage2_imag[1] <= stage1_imag[1] + stage1_imag[5];
        stage2_real[2] <= stage1_real[2] + stage1_real[6];
        stage2_imag[2] <= stage1_imag[2] + stage1_imag[6];
        stage2_real[3] <= stage1_real[3] + stage1_real[7];
        stage2_imag[3] <= stage1_imag[3] + stage1_imag[7];
        stage2_real[4] <= stage1_real[0] - stage1_real[4]; // Subtract for
            alternate branch
        stage2_imag[4] <= stage1_imag[0] - stage1_imag[4];
        stage2_real[5] <= stage1_real[1] - stage1_real[5];
        stage2_imag[5] <= stage1_imag[1] - stage1_imag[5];
        stage2_real[6] <= stage1_real[2] - stage1_real[6];
        stage2_imag[6] <= stage1_imag[2] - stage1_imag[6];
        stage2_real[7] <= stage1_real[3] - stage1_real[7];
        stage2_imag[7] <= stage1_imag[3] - stage1_imag[7];
        end
        end
        endmodule
```

Listing 4: Registers for FFT Stages

10.4.3 Butterfly Operations

The butterfly operation performs the core FFT computation:

$$X[k] = x_1 + W_N^k \cdot x_2,$$
$$X[k + N/2] = x_1 - W_N^k \cdot x_2,$$

where W_N^k is the twiddle factor. Below is the implementation in Verilog:

```
        module butterfly (
        input wire signed [15:0] x1_real,
        input wire signed [15:0] x1_imag,
        input wire signed [15:0] x2_real,
        input wire signed [15:0] x2_imag,
        input wire signed [15:0] twiddle_real,
        input wire signed [15:0] twiddle_imag,
        output wire signed [15:0] out1_real,
        output wire signed [15:0] out1_imag,
        output wire signed [15:0] out2_real,
        output wire signed [15:0] out2_imag
        );
        wire signed [31:0] temp1_real, temp1_imag;
```

```
assign temp1_real = twiddle_real * x2_real - twiddle_imag * x2_imag;
assign temp1_imag = twiddle_real * x2_imag + twiddle_imag * x2_real;

assign out1_real = x1_real + temp1_real[31:16];
assign out1_imag = x1_imag + temp1_imag[31:16];
assign out2_real = x1_real - temp1_real[31:16];
assign out2_imag = x1_imag - temp1_imag[31:16];
endmodule
```

Listing 5: Butterfly Operation

10.4.4 Twiddle Factors

The twiddle factors are constants used to multiply the inputs during butterfly operations. These are stored in a ROM and precomputed as shown below:

```
module twiddle_rom (
output reg signed [15:0] twiddle_real [0:3],
output reg signed [15:0] twiddle_imag [0:3]
);
initial begin
twiddle_real[0] = 16'h4000; // W_8^0 = 1
twiddle_imag[0] = 16'h0000; // Imaginary = 0
twiddle_real[1] = 16'h2D41; // W_8^1 = cos(-pi/4)
twiddle_imag[1] = 16'hD8E4; // Imaginary = sin(-pi/4)
twiddle_real[2] = 16'h0000; // W_8^2 = 0
twiddle_imag[2] = 16'hC000; // Imaginary = -1
twiddle_real[3] = 16'hD8E4; // W_8^3 = cos(-3pi/4)
twiddle_imag[3] = 16'h2D41; // Imaginary = sin(-3pi/4)
end
endmodule
```

Listing 6: Twiddle Factors

10.4.5 FSM for FFT Stages

The FFT computation progresses sequentially through the stages, controlled by a finite state machine (FSM). The FSM moves through the stages as follows:

```
module fft_fsm (
input wire clk,
input wire rst,
output reg [1:0] stage
);
always @(posedge clk or negedge rst) begin
if (!rst)
stage <= 2'b00; // Reset to stage 0
else
stage <= stage + 1; // Move to the next stage
end
endmodule
```

Listing 7: FSM for FFT Stages

10.4.6 Top Module for FFT System

The top module integrates all child modules (FSM, Twiddle ROM, FFT Registers, and Butterfly modules) to perform FFT computations. Below is the Verilog code for the top module:

```
module fft_top (
input wire clk,
input wire rst,
input wire [15:0] in_real,
input wire [15:0] in_imag,
output wire [15:0] out_real,
output wire [15:0] out_imag
);
// Intermediate signals
wire [15:0] stage1_real[0:7], stage1_imag[0:7];
wire [15:0] stage2_real[0:7], stage2_imag[0:7];
wire signed [15:0] twiddle_real[0:3], twiddle_imag[0:3];
reg [1:0] current_stage;

// Instantiate FSM to control stages
fft_fsm fsm (
.clk(clk),
.rst(rst),
.stage(current_stage)
);

// Instantiate twiddle factor ROM
twiddle_rom twiddle_rom_inst (
.twiddle_real(twiddle_real),
.twiddle_imag(twiddle_imag)
);

// Instantiate FFT registers
fft_registers fft_registers_inst (
.clk(clk),
.rst(rst),
.in_real(in_real),
.in_imag(in_imag),
.stage1_real(stage1_real),
.stage1_imag(stage1_imag),
.stage2_real(stage2_real),
.stage2_imag(stage2_imag)
);

// Instantiate butterfly modules for each stage
generate
genvar i;
for (i = 0; i < 4; i = i + 1) begin : butterfly_stage1
butterfly butterfly_inst (
.x1_real(stage1_real[i]),
.x1_imag(stage1_imag[i]),
.x2_real(stage1_real[i + 4]),
.x2_imag(stage1_imag[i + 4]),
.twiddle_real(twiddle_real[i]),
.twiddle_imag(twiddle_imag[i]),
.out1_real(stage2_real[i]),
```

```
        .out1_imag(stage2_imag[i]),
        .out2_real(stage2_real[i + 4]),
        .out2_imag(stage2_imag[i + 4])
        );
        end
        endgenerate

        // Output assignment
        assign out_real = stage2_real[0]; // Final output real value
        assign out_imag = stage2_imag[0]; // Final output imaginary value
        endmodule
```

Listing 8: Top Module for FFT System

11 Conclusion

The 8-point FFT implementation in Verilog involves systematic processing across three stages using bit-reversed input, butterfly operations, precomputed twiddle factors, and pipelining. This design is scalable and forms the foundation for larger FFT implementations. The example illustrates the process and highlights how hardware can efficiently perform complex computations in signal processing.