Project Report : 4-Point FFT

Mahendra Khinchi

Mathematical Derivation of 4-point Radix-2 FFT

The Fast Fourier Transform (FFT) is an efficient algorithm to compute the Discrete Fourier Transform (DFT) with reduced computational complexity. For an N-point DFT, the FFT reduces the complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log_2 N)$.

In this experiment, we implement a 4-point Radix-2 Decimation-in-Time (DIT) FFT. Given a complex input sequence:

$$x[0],\ x[1],\ x[2],\ x[3]$$

where each $x[n]$ is of the form $x[n] = a[n] + jb[n]$, the DFT is defined as:

$$X[k] = \sum_{n=0}^{3} x[n] \cdot W_4^{nk}, \quad \text{for } k = 0, 1, 2, 3$$

where $W_4 = e^{-j\frac{2\pi}{4}} = -j$ is the twiddle factor.

The twiddle factors used in the 4-point FFT are:

$$W_4^0 = 1, \quad W_4^1 = -j, \quad W_4^2 = -1, \quad W_4^3 = j$$

Stage 1: Butterfly Computations

We divide the input into even and odd indices:

$$\begin{aligned} A_0 &= x[0] + x[2] \\ A_1 &= x[0] - x[2] \\ B_0 &= x[1] + x[3] \\ B_1 &= (x[1] - x[3]) \cdot W \end{aligned}$$

For $N = 4$, the twiddle factor $W = -j$ is used for B_1.

Stage 2: Final Outputs

$$\begin{aligned} X[0] &= A_0 + B_0 \\ X[1] &= A_1 + B_1 \\ X[2] &= A_0 - B_0 \\ X[3] &= A_1 - B_1 \end{aligned}$$

Verilog Code for 4-point Streaming FFT

Design Code

```
// Code your design here
module fft4_streaming (
    input clk,
    input rst,
    input valid_in,
    input signed [7:0] real_in,
    input signed [7:0] imag_in,
    output reg valid_out,
    output reg signed [15:0] real_out,
    output reg signed [15:0] imag_out
);

    reg signed [7:0] real_buffer[0:3];
    reg signed [7:0] imag_buffer[0:3];
    reg [1:0] sample_count;
    reg [1:0] output_index;
    reg [2:0] state;

    reg signed [15:0] real_tmp[0:3];
    reg signed [15:0] imag_tmp[0:3];

    reg signed [15:0] a0r, a0i, a1r, a1i, a2r, a2i, a3r, a3i;
    reg signed [15:0] b0r, b0i, b1r, b1i, b2r, b2i, b3r, b3i;

    localparam IDLE = 3'd0,
               COLLECT = 3'd1,
               COMPUTE = 3'd2,
               OUTPUT = 3'd3;

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            sample_count <= 0;
            output_index <= 0;
            state <= IDLE;
            valid_out <= 0;
        end else begin
            case (state)
                IDLE: begin
                    if (valid_in) begin
                        real_buffer[0] <= real_in;
                        imag_buffer[0] <= imag_in;
                        sample_count <= 1;
                        state <= COLLECT;
                    end
                    valid_out <= 0;
                end

                COLLECT: begin
                    if (valid_in) begin
                        real_buffer[sample_count] <= real_in;
                        imag_buffer[sample_count] <= imag_in;
                        sample_count <= sample_count + 1;
                        if (sample_count == 2'd3)
```

```
                            state <= COMPUTE;
                        end
                        valid_out <= 0;
                    end

                    COMPUTE: begin
                        a0r = real_buffer[0]; a0i = imag_buffer[0];
                        a1r = real_buffer[1]; a1i = imag_buffer[1];
                        a2r = real_buffer[2]; a2i = imag_buffer[2];
                        a3r = real_buffer[3]; a3i = imag_buffer[3];

                        b0r = a0r + a2r; b0i = a0i + a2i;
                        b1r = a1r + a3r; b1i = a1i + a3i;
                        b2r = a0r - a2r; b2i = a0i - a2i;
                        b3r = a1i - a3i; b3i = a3r - a1r;

                        real_tmp[0] <= b0r + b1r;
                        imag_tmp[0] <= b0i + b1i;

                        real_tmp[1] <= b2r + b3r;
                        imag_tmp[1] <= b2i + b3i;

                        real_tmp[2] <= b0r - b1r;
                        imag_tmp[2] <= b0i - b1i;

                        real_tmp[3] <= b2r - b3r;
                        imag_tmp[3] <= b2i - b3i;

                        output_index <= 0;
                        state <= OUTPUT;
                        valid_out <= 0;
                    end

                    OUTPUT: begin
                        valid_out <= 1;
                        real_out <= real_tmp[output_index];
                        imag_out <= imag_tmp[output_index];
                        output_index <= output_index + 1;
                        if (output_index == 2'd3)
                            state <= IDLE;
                    end
                endcase
            end
        end
    endmodule
```

Testbench Code

```
// Code your testbench here
// or browse Examples
`timescale 1ns / 1ps

module tb_fft4_streaming;

    reg clk;
```

```
    reg rst;
    reg valid_in;
    reg signed [7:0] real_in;
    reg signed [7:0] imag_in;
    wire valid_out;
    wire signed [15:0] real_out;
    wire signed [15:0] imag_out;

    fft4_streaming uut (
        .clk(clk),
        .rst(rst),
        .valid_in(valid_in),
        .real_in(real_in),
        .imag_in(imag_in),
        .valid_out(valid_out),
        .real_out(real_out),
        .imag_out(imag_out)
    );

    // Clock
    always #5 clk = ~clk;

    // Input samples
    reg signed [7:0] real_samples[0:3];
    reg signed [7:0] imag_samples[0:3];

    integer i;

    initial begin
        // VCD setup for EDA Playground
        $dumpfile("fft4_streaming.vcd");
        $dumpvars(1, uut); // Limit scope to DUT only

        // Init
        clk = 0;
        rst = 1;
        valid_in = 0;
        real_in = 0;
        imag_in = 0;

        // Sample input
        real_samples[0] = 8'd10; imag_samples[0] = 8'd0;
        real_samples[1] = 8'd20; imag_samples[1] = 8'd0;
        real_samples[2] = 8'd30; imag_samples[2] = 8'd0;
        real_samples[3] = 8'd40; imag_samples[3] = 8'd0;

        #20;
        rst = 0;

        // Feed inputs
        for (i = 0; i < 4; i = i + 1) begin
            @(posedge clk);
            valid_in <= 1;
            real_in <= real_samples[i];
            imag_in <= imag_samples[i];
        end

```

```
        @(posedge clk);
        valid_in <= 0;

        // Wait for output
        wait (valid_out);
        for (i = 0; i < 4; i = i + 1) begin
            @(posedge clk);
            if (valid_out)
                $display("FFT[%0d] = %d + j%d", i, real_out,
                    imag_out);
        end

        #20;
        $finish;
    end

endmodule
```

MATLAB Verification

```
real_in = [10, 20, 30, 40];
imag_in = [0, 0, 0, 0];

x = real_in + 1j * imag_in;
X = fft(x);

disp('FFT Output:');
disp(X);
```

MATLAB Output

The MATLAB FFT output for the input `[10, 20, 30, 40]` (with imaginary part zero) is:

$$X[0] = 100 + 0j$$
$$X[1] = 20 + j$$
$$X[2] = -20 + j$$
$$X[3] = -20 + j$$

This matches the Verilog simulation output in both magnitude and phase, confirming the correctness of the hardware implementation.