## `E0 217 Project Report (Aug 2023)

**Project Title:** Design, Implementation and Simulation of an 8-point FFT Circuit in 45nm CMOS

TABLE OF CONTENTS:

|  |  |  |
| --- | --- | --- |
| **S.NO** | **TITLE** | **PAGE NUMBER** |
| 1 | Introduction about group members and individual Contributions | 2 |
| 2 | Number of clock cycles for FFT computation | 2 |
| 3 | Area of synthesized design | 3 |
| 4 | Analysis of max clock frequency | 3 |
| 5 | Power Consumption analysis | 5 |
| 6 | Energy Consumption analysis | 6 |
| 7 | Error Computation | 6 |
| 8 | Modification of FFT circuit to compute inverse FFT | 8 |
| 9 | Design Details and explanation of implementation | 8 |
| 10 | Architectural Trade-offs | 11 |
| 11 | References | 11 |

**E0 217 Project Report (Aug 2023)**

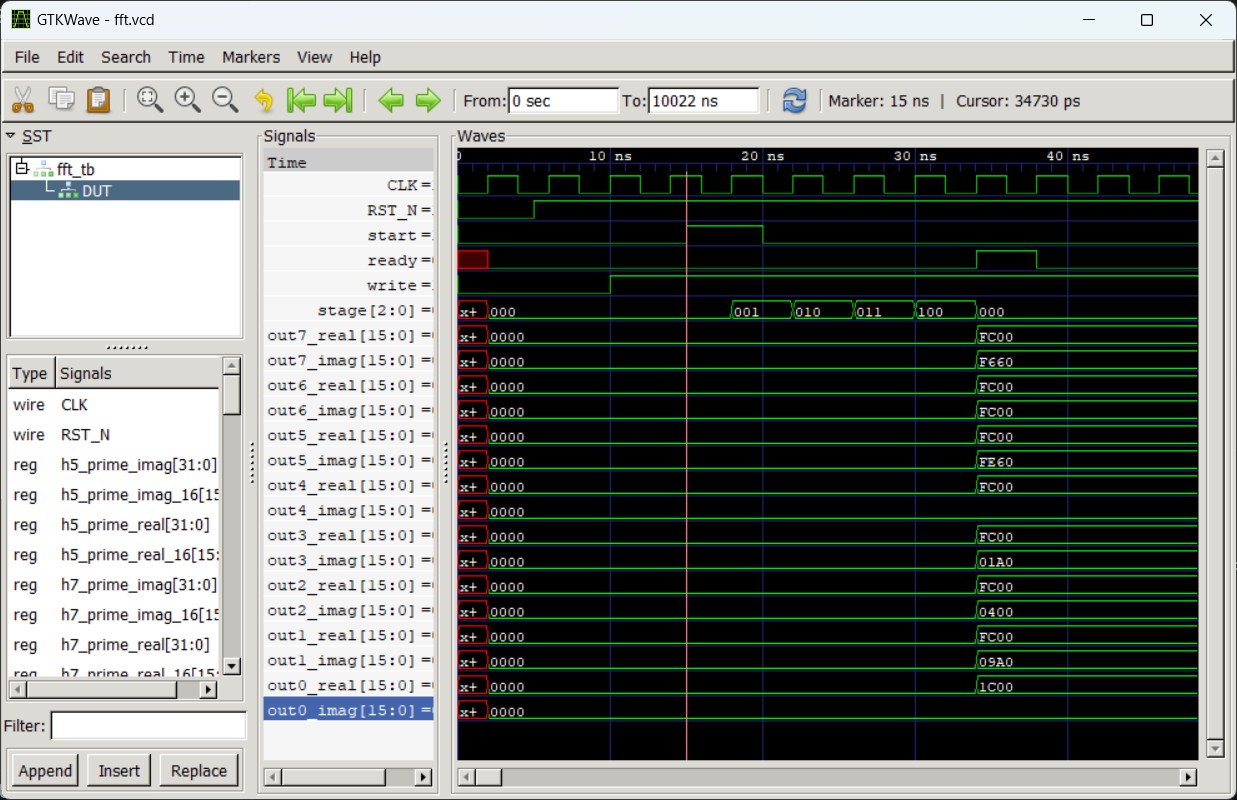
**Names and IISc Email IDs of Group Members:**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Name** | **SR No.** | **IISc Email ID** |
| 1. | **Guhan Rajasekar** | **22410** | [**guhanr@iisc.ac.in**](mailto:guhanr@iisc.ac.in) |
| 2. | **Ujjwal Chaudhary** | **22577** | [**ujjwalc@iisc.ac.in**](mailto:ujjwalc@iisc.ac.in) |

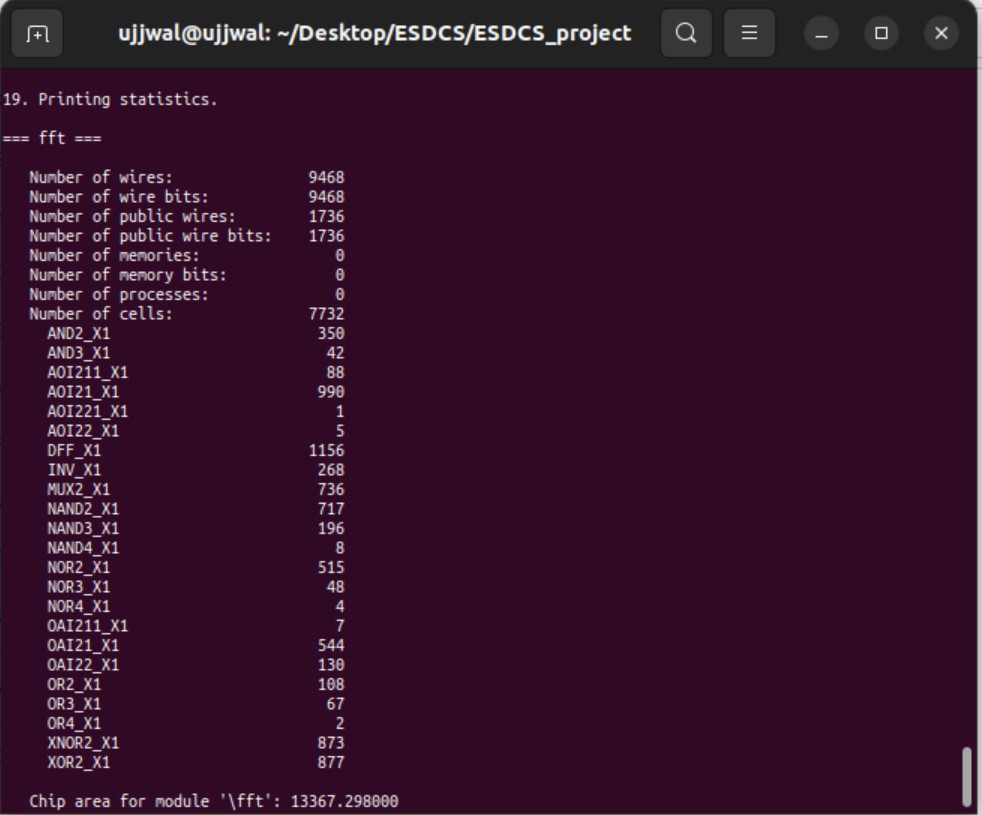
**Individual Contributions**

|  |  |  |
| --- | --- | --- |
|  | **Name** | **Contributions** |
| 1. | **Guhan Rajasekar** | * Implementation of FFT code (without control signals) * Error computation between obtained and desired results * Report compilation |
| 2. | **Ujjwal Chaudhary** | * Optimization of code (changing combinational blocks to sequential blocks) and making the code compatible with CLK signal and control signals like RST, WRITE, START, READY * Creation of test bench and simulation of design in GTK Wave. * Performed area, power and speed analysis. |

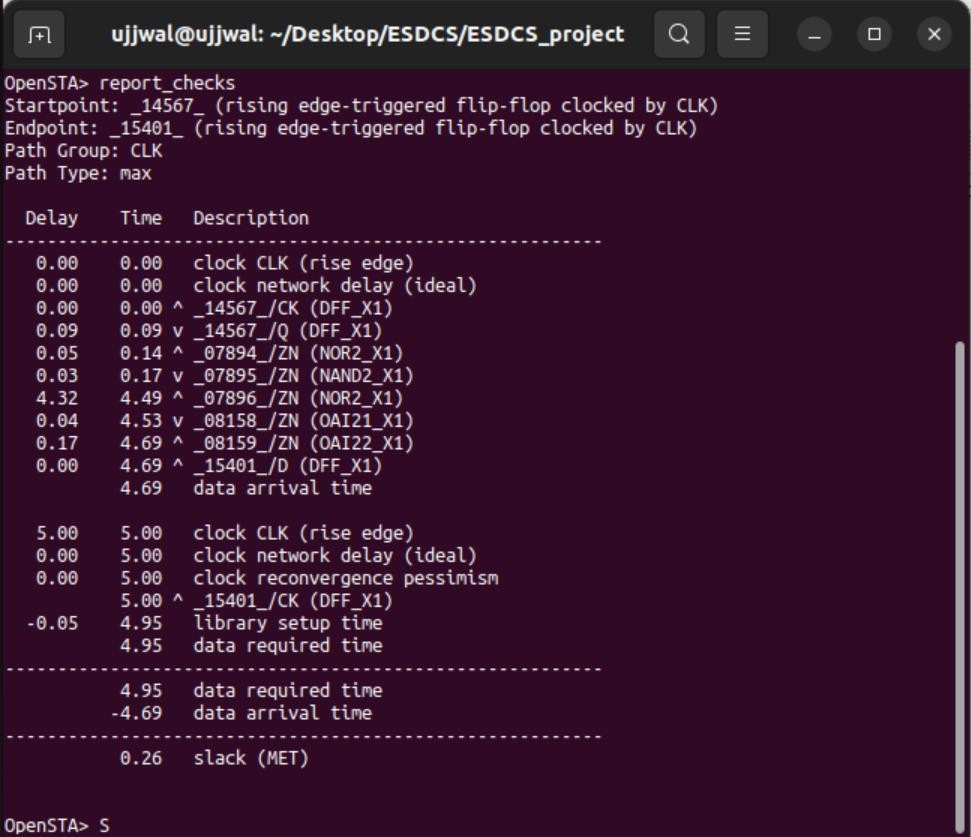
**Number of clock cycles per FFT computation:**

* Number of clock cycles between beginning of start and beginning of ready signal = **4.75**
* 

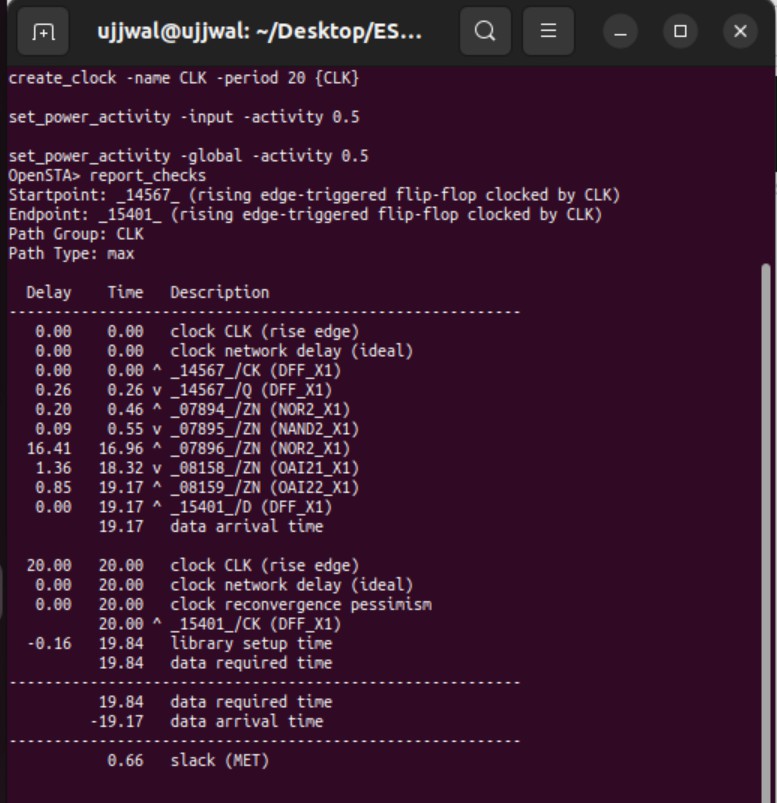
## Area of synthesized design:

* 
* Area of synthesized design is **13367.298 µm2**.

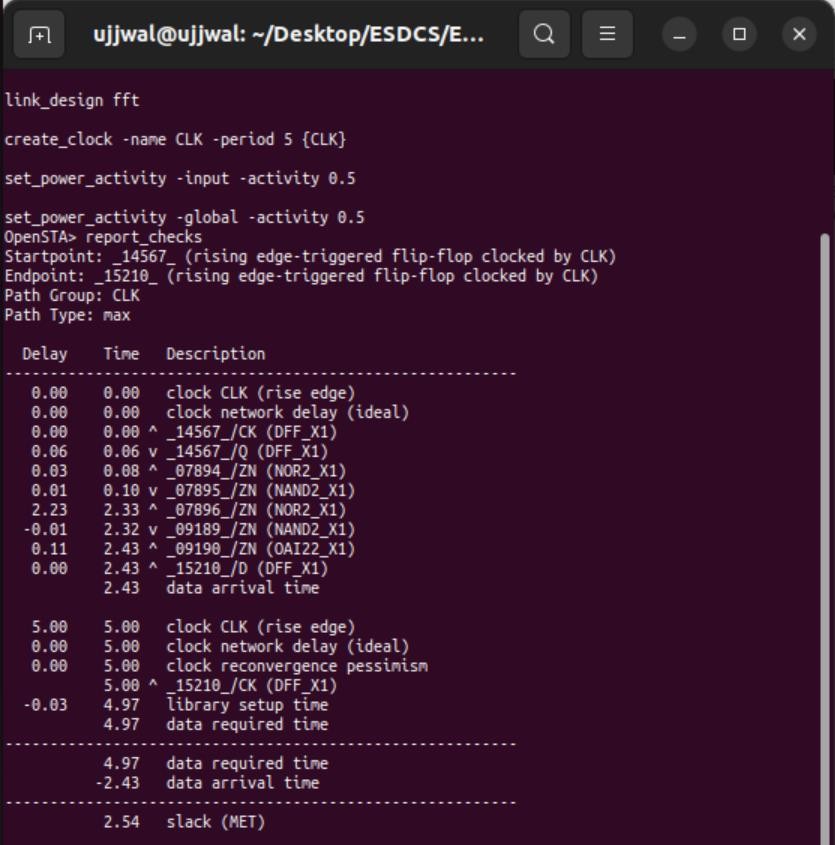
## Max clock frequency supported by the synthesized design:

* **Typical corner, 25**℃ **and 1V**
  + 
  + This result is for typical corner, 25℃ and 1V.
  + Input clock signal period is 5ns.
  + Here the slack period is 0.26ns.
  + Supported clock period is 5ns – 0.26ns = **4.74ns**
  + Operating frequency is **210.97 MHz**.

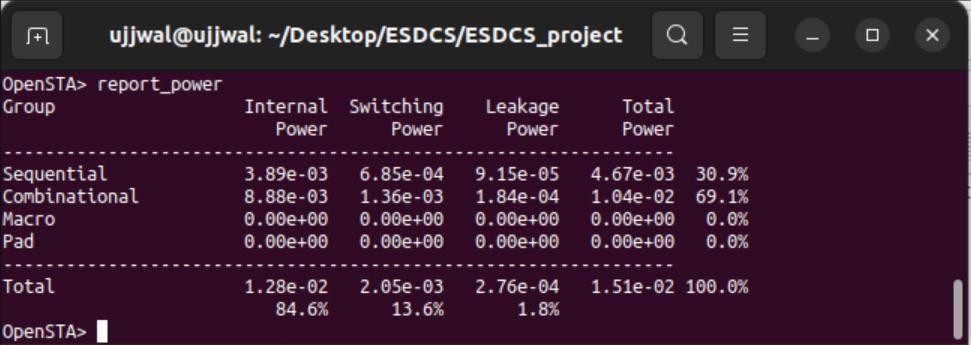
## SS corner, 25℃, 1V

* + 
  + This is for SS corner, 25℃ and 1V.
  + Input clock period is 20ns. (because this case does not support clock signal of 5ns)
  + Slack period is 0.66 ns.
  + Supported clock signal period is 20ns – 0.66ns = **19.34ns**
  + Operating frequency = **51.706MHz.**

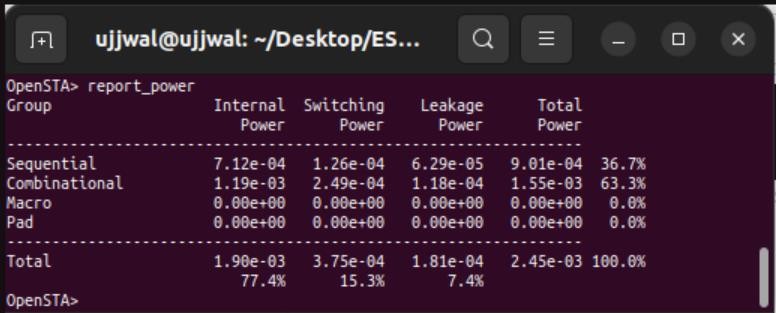
## FF corner, 25℃, 1V

* + 
  + Input clock signal period = 5ns.
  + Slack period = 2.54 ns.
  + Clock period supported by the design = **2.46ns.**
  + Max operating frequency at FF corner = **406.504MHz**

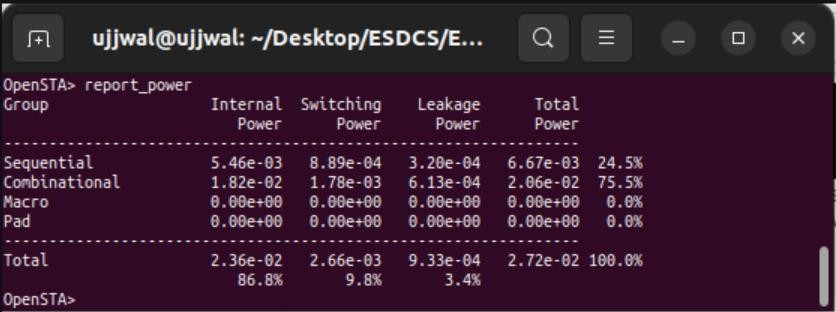
## Power Consumption of the synthesized design:

* **Power Consumption in typical corner, 25**℃ **and 1V:**
  + 
  + The total power consumed at typical corner, 25℃ and 1V is **15.1mW**.

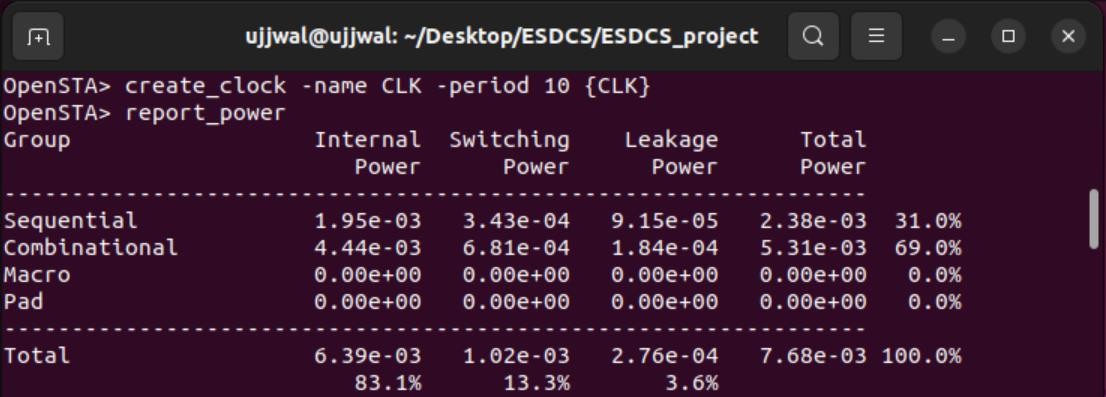
## Power Consumption in SS corner,25℃ and 1V.

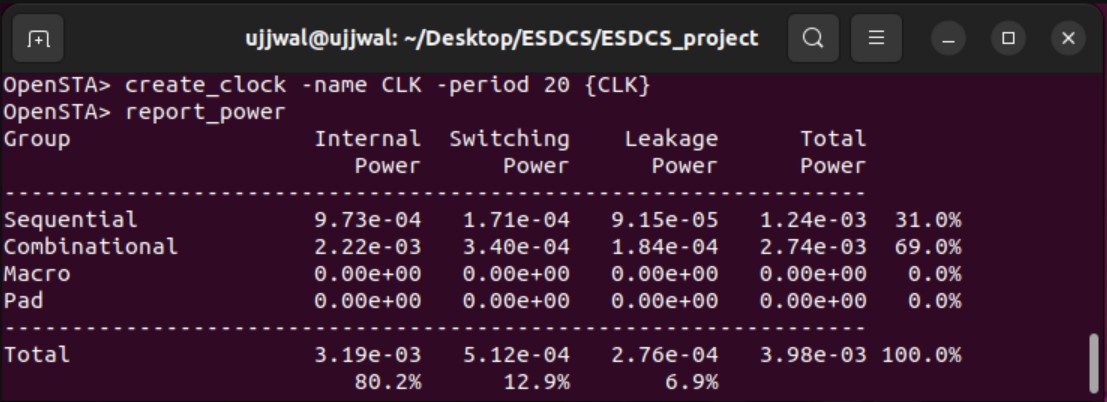
* + 
  + Power consumed at SS corner, 25℃ and 1V is **2.45mW**

## Power Consumption in FF corner,25℃ and 1V.

* + 
  + Total power consumption at FF corner, 25℃ and 1V is **27.2mW**.

## Energy Consumption of synthesized design:

* Typical, 25℃, 1V, T=5ns (Image attached above)
* Typical, 25℃, 1V, T=10ns
* Typical, 25℃, 1V, T=20ns



|  |  |  |  |
| --- | --- | --- | --- |
| **Corner Case** | **Power Consumption (mW)** | **Frequency (MHz)** | **Energy Consumption (pJ)** |
| Typical, 25℃, 1V | 15.1 | 200 | 75.5 |
| Typical, 25℃, 1V | 7.68 | 100 | 76.8 |
| Typical, 25℃, 1V | 3.98 | 50 | 79.6 |

## Dependence of Energy Consumption of the synthesized design on the clock frequency:

* From above mentioned table, we conclude that **energy consumption reduces as clock frequency increases.**

## Maximum absolute error when computing FFT of {0, 1, 2, 3, 4, 5, 6, 7}

* Let the 8-point FFT result be denoted as X(0), X(1), X(2), X(3), X(4), X(5), X(6) and X(7)
* Here a **decimal point has been added in the obtained results to indicate there are 8 bits for the fractional part, 7 bits for the decimal part and MSB is the sign bit.** This holds true for both real and imaginary parts.
* In the following calculations, √2 is considered as **1.414**.

# X(0) :

* + Expected = 28 + j0
  + Obtained = (1c.00 + j00.00)H = 28 + j0
  + Absolute Error = 0

# X(1)

* + Expected = -4.0 + j 9.656
  + Obtained = (fc.00 + j09.a0)H = -4 + j9.625
  + Absolute Error = |(-4.0 + j 9.656) – (-4 + j9.625) | = 0.031

# X(2)

* + Expected = -4 + j4
  + Obtained = (fc.00 + j04.00)H = (-4 + j4)
  + Absolute Error = 0

# X(3)

* + Expected = -4 + j1.656
  + Obtained = (fc.00 + j01.a0)H = -4 + j1.625
  + Absolute Error = |(-4 + j1.656) – (-4 + j1.625) | = 0.031

# X(4)

* + Expected = -4 + j0
  + Obtained = (fc.00 + j00.00)H = -4 + j0
  + Absolute Error = 0

# X(5)

* + Expected = -4 – j1.656
  + Obtained = (fc.00 + jfe.60)H = (-4 – j1.402)
  + Absolute Error = |(-4 – j1.656 ) – (-4 – j1.402) | = 0.254

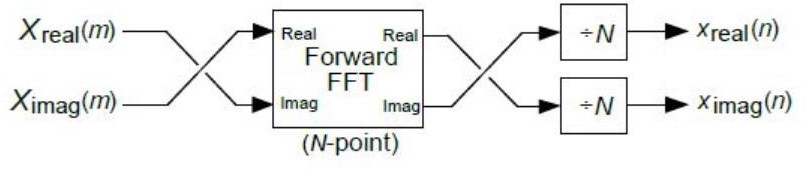
# X(6)

* + Expected = -4.0 – j4.0
  + Obtained = (fc.00 + jfc.00)H = -4 - j4
  + Absolute Error = 0

# X(7)

* + Expected = -4 – j9.656
  + Obtained = (fc.00 + jf6.60)H = -4 – j9.625
  + Absolute Error = | (-4 – j9.656 ) – (-4 – j9.625) | = 0.031

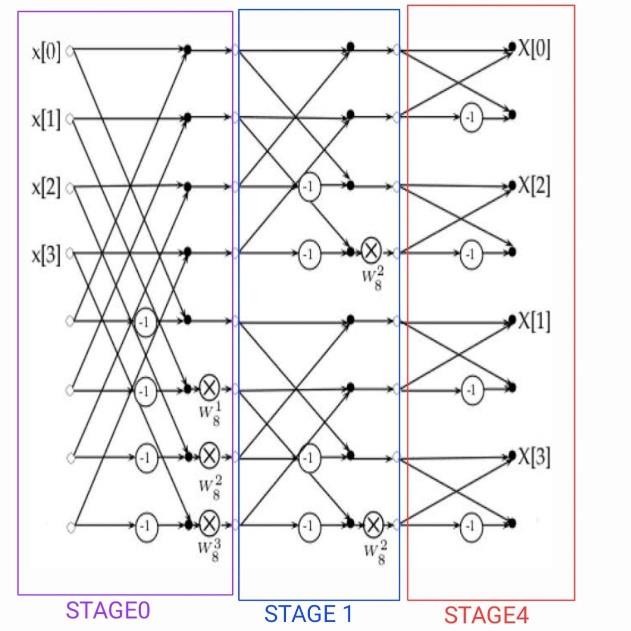
## Modifying the fft circuit to compute inverse fft

* Inverse fft can be computed using Data Swapping technique, as shown below:
* ​
* The real part of FFT input is fed as imaginary input to the circuit and the imaginary part of

FFT is fed as real input to the circuit.

* The real portion of the output gives imaginary part of expected time domain sequence scaled by N and the imaginary part of the output gives the real part of the time domain sequence scaled by N. To get the desired outputs, we divide them by N.
* So, in addition to the existing hardware circuitry, extra circuitry will be required to scale by factor of (1/N) to get the desired time domain inverse FFT sequence.

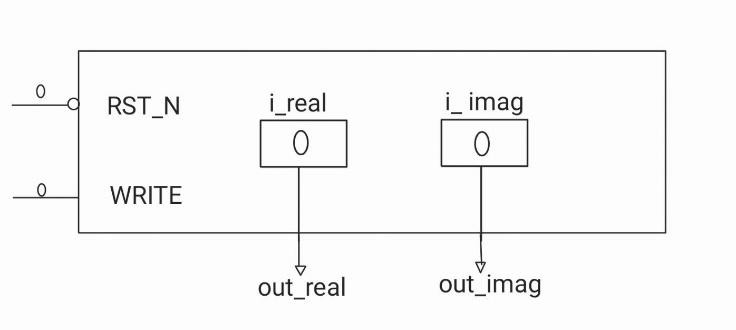
## Design Details:

* The entire FFT computation has been divided into five stages as follows:
* 
* Stage 2 and Stage 3 are used for intermediate variable computations in the code.
* But the overall FFT computation can be broadly classified into stages 0, 1 and 4.
* If the RST\_N signal is low, the value 0 is written in the registers i0\_real, i0\_imag,

i1\_real,i1\_imag, …… , i7\_real, i7\_imag.

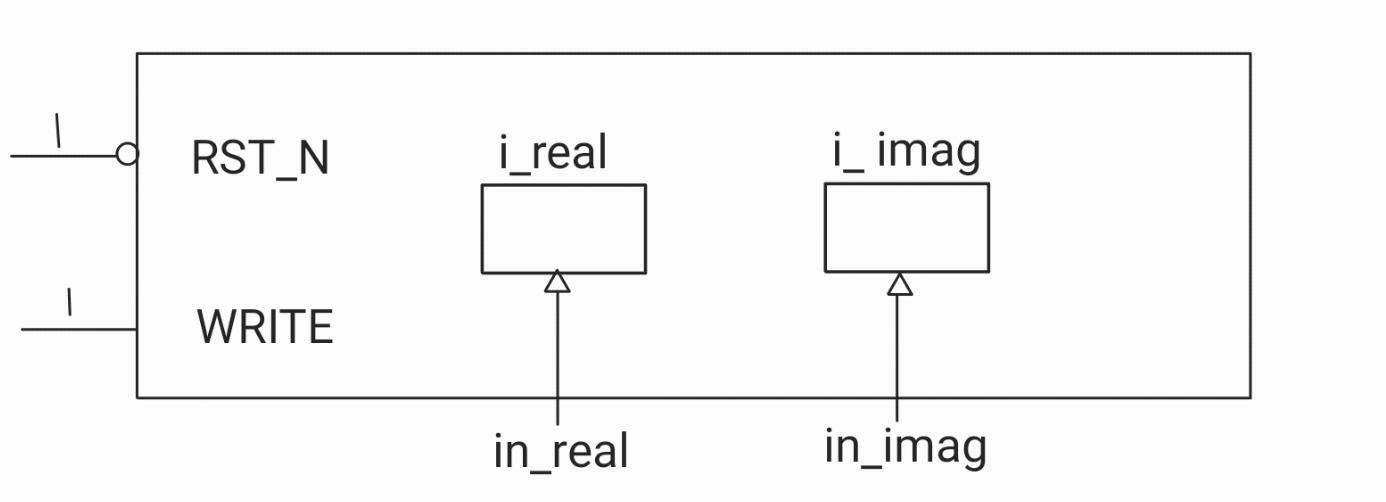
* The output wires out0\_real,out0\_imag, out1\_real,out1\_imag,…..out7\_real,out7\_imag are

continuously driven by the data present in the registers as shown below.

* 
* In the above block, although only two registers are shown, there are total of 16 registers ( 8 for the real part and 8 for the imaginary part ) and each register holds 16 bit data.

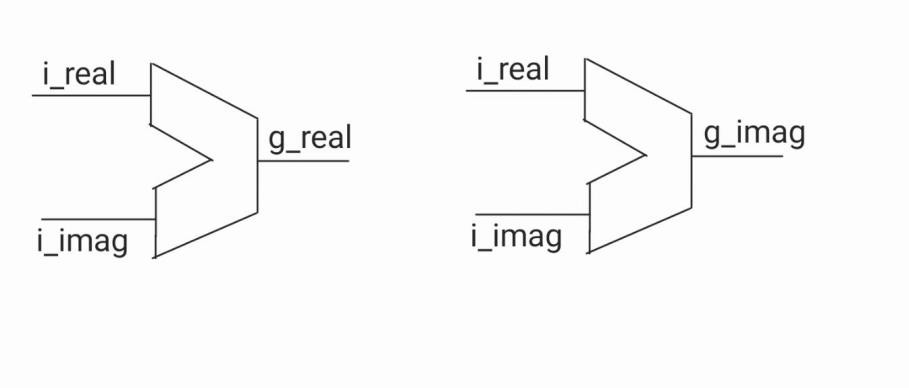
## Storing Inputs in registers:

* + The circuit stores inputs in registers when **write** signal is high.
  + When write signal is high, the registers i0\_real ,i0\_imag, i1\_real,i1\_imag,… ,i7\_real, i7\_imag will store the value held by the continuously driven wires in0\_real,in0\_imag,in1\_real,in1\_imag…..,in7\_real,in7\_imag respectively as shown below:

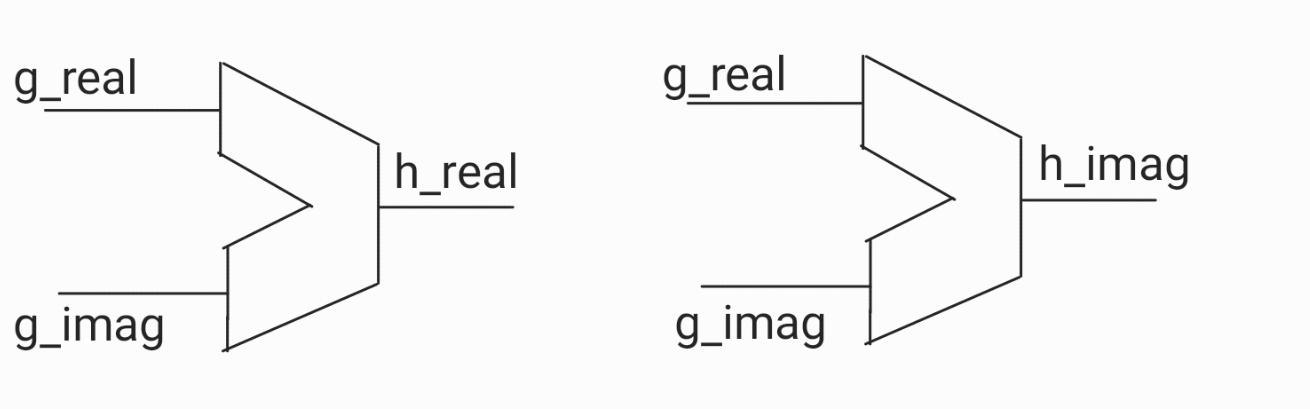


* + Here as well only two signals in\_real and in\_imag are shown for brevity of block diagram.
  + But there are 16 wires ( 8 for real and 8 for imaginary parts) that are continuously driven.
* Computation commences when **start** signal is high.
* The value stored by 2-bit register variable named **stage** indicates the stage of FFT computation.

## Stage 0 computations:

* + The contents in i0\_real, i0\_imag,…….,i7\_real,i7\_imag act as inputs for stage 0 computations.
  + The registers g\_real[0], g\_imag[0], g\_real[1],g\_imag[1]…..,g\_real[7],g\_imag[7] store the results of stage 0.
  + 
  + Once again for brevity, not all the registers of stage 0 are shown.
  + There are 16 registers. 8 registers hold the real part of stage 0 results and 8 registers hold the imaginary part of stage 0 results .
  + Stage 0 computations is based on the formula obtained from the butterfly diagram.
  + Once stage 0 computations are done, the content of **stage register** is incremented by 1 to go to the next stage.

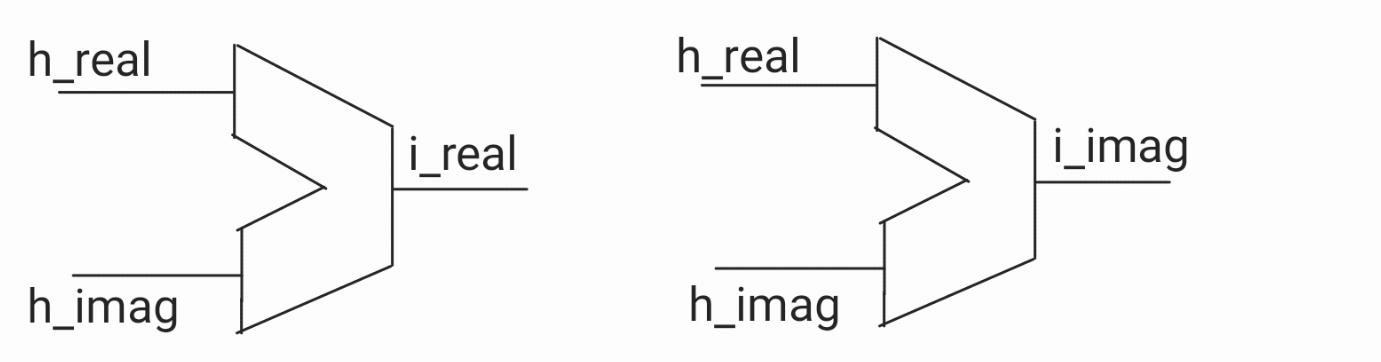
## Stage 1 computations

* + For stage 1 computations, the contents of the registers g\_real[0], g\_imag[0], g\_real[1],g\_imag[1]…..,g\_real[7],g\_imag[7] will be the inputs and the results are stored in the registers h\_real[0],h\_real[1],…..,h\_real[7],h\_imag[0],……,h\_imag[7]
  + 
  + The computations are based on the equations obtained from the butterfly diagram.

## Stage 2 computations

* + In this stage , the registers temp0, temp1 ,temp2 and temp3 are populated as follows:
    - temp0 = h\_real[5] + h\_imag[5]
    - temp1 = h\_imag[5] - h\_real[5]
    - temp2 = h\_imag[7] - h\_real[7]
    - temp3 = -(h\_imag[7] + h\_real[7])
  + These calculations aid in the computation of the final result.
  + Once the above calculations are done, contents of the stage register is incremented by 1 and we go to the next stage.

## Stage 3 computations

* + In this stage, the contents of temp0, temp1, temp2, temp3 are multiplied with 0.707 ( hard coded in binary in the code) .
  + Then content of the stage register is incremented and we go to the final stage.
* **Stage 4 computations:**
  + Here the contents of h\_real[0],h\_imag[0],h\_real[1],h\_imag[1]…,h\_real[7],h\_imag[7] act as inputs and the final output is stored in the registers i\_real[0],i\_imag[0],i\_real[1],i\_imag[1],…i\_real[7],i\_imag[7].
  + 
  + The wires out0\_real , out0\_imag, out1\_real, out1\_imag,…….out7\_real,out7\_imag are continuously driven by the contents in these registers. Hence at any point of time, the output at that instant is present in these output wires.

## Architectural Trade-offs

* In this FFT implementation, focus is low latency.
* Hence the inbuilt multiplication function has been used with the ‘ \* ‘ symbol to get faster

multiplication results that are required at the intermediate stages.

* This comes at the cost of more area and power consumption.
* As an addition, pipelining can be implemented if we want to improve throughput.

**Verilog Code to implement fft:**

module fft(CLK, RST\_N, write, start, in0\_real , in0\_imag, in1\_real, in1\_imag,

 in2\_real  ,in2\_imag, in3\_real  ,in3\_imag, in4\_real  ,in4\_imag, in5\_real  ,in5\_imag,

 in6\_real  ,in6\_imag, in7\_real  ,in7\_imag, ready, out0\_real ,out0\_imag, out1\_real,

 out1\_imag, out2\_real ,out2\_imag, out3\_real ,out3\_imag, out4\_real ,out4\_imag,

 out5\_real ,out5\_imag, out6\_real ,out6\_imag, out7\_real ,out7\_imag);

  input CLK, RST\_N;

  input write, start;

  input wire signed [15:0]  in0\_real  ,in0\_imag;

  input wire signed [15:0]  in1\_real  ,in1\_imag;

  input wire signed [15:0]  in2\_real  ,in2\_imag;

  input wire signed [15:0]  in3\_real  ,in3\_imag;

  input wire signed [15:0]  in4\_real  ,in4\_imag;

  input wire signed [15:0]  in5\_real  ,in5\_imag;

  input wire signed [15:0]  in6\_real  ,in6\_imag;

  input wire signed [15:0]  in7\_real  ,in7\_imag;

  output reg ready;

  output wire signed [15:0] out0\_real ,out0\_imag;

  output wire signed [15:0] out1\_real ,out1\_imag;

  output wire signed [15:0] out2\_real ,out2\_imag;

  output wire signed [15:0] out3\_real ,out3\_imag;

  output wire signed [15:0] out4\_real ,out4\_imag;

  output wire signed [15:0] out5\_real ,out5\_imag;

  output wire signed [15:0] out6\_real ,out6\_imag;

  output wire signed [15:0] out7\_real ,out7\_imag;

  reg signed [15:0] i0\_real, i0\_imag;

  reg signed [15:0] i1\_real, i1\_imag;

  reg signed [15:0] i2\_real, i2\_imag;

  reg signed [15:0] i3\_real, i3\_imag;

  reg signed [15:0] i4\_real, i4\_imag;

  reg signed [15:0] i5\_real, i5\_imag;

  reg signed [15:0] i6\_real, i6\_imag;

  reg signed [15:0] i7\_real, i7\_imag;

  reg [2:0] stage;

  reg signed [15:0] g\_real [0:7];   // Declare g\_real as an array of registers to store real part of stage 1 results

  reg signed [15:0] g\_imag [0:7];   // Declare g\_imag as an array of registers to store imag part of stage 1 results

  reg signed [15:0] h\_real [0:7];   // Declare h\_real as an array of registers to store real part of stage 2 results

  reg signed [15:0] h\_imag [0:7];   // Declare h\_imag as an array of registers to store imag part of stage 2 results

  reg signed [15:0] i\_real [0:7];   // Declare i\_real as an array of registers to store real part of stage 3 results

  reg signed [15:0] i\_imag [0:7];   // Declare i\_imag as an array of registers to store imag part of stage 3 results

  reg signed [15:0] temp1,temp2,temp3,temp0;

  reg signed [31:0] h5\_prime\_real;  // 32 bit register to store real part of h[5] \* (W8 ^ 1)

  reg signed [31:0] h5\_prime\_imag;  // 32 bit register to store imag part of h[5] \* (W8 ^ 1)

  reg signed [31:0] h7\_prime\_real;  // 32 bit register to store real part of h[7] \* (W8 ^ 3)

  reg signed [31:0] h7\_prime\_imag;  // 32 bit register to store imag part of h[7] \* (W8 ^ 3)

  reg signed [15:0] h5\_prime\_real\_16;  // 16 bit register to store real part of h[5] \* (W8 ^ 1)

  reg signed [15:0] h5\_prime\_imag\_16;  // 16 bit register to store imag part of h[5] \* (W8 ^ 1)

  reg signed [15:0] h7\_prime\_real\_16;  // 16 bit register to store real part of h[7] \* (W8 ^ 3)

  reg signed [15:0] h7\_prime\_imag\_16;  // 16 bit register to store imag part of h[7] \* (W8 ^ 3)

  always @(posedge CLK) begin

    if (~RST\_N) begin

      ready <= 1'b0;

      stage <= 2'b00;

      i\_real[0] <= 0; i\_imag[0]<=0;

      i\_real[1] <= 0; i\_imag[1]<=0;

      i\_real[2] <= 0; i\_imag[2]<=0;

      i\_real[3] <= 0; i\_imag[3]<=0;

      i\_real[4] <= 0; i\_imag[4]<=0;

      i\_real[5] <= 0; i\_imag[5]<=0;

      i\_real[6] <= 0; i\_imag[6]<=0;

      i\_real[7] <= 0; i\_imag[7]<=0;

    end

    else begin

      if (start) begin

        g\_real[0] <= i0\_real + i4\_real;

        g\_imag[0] <= i0\_imag + i4\_imag;

        g\_real[1] <= i0\_real - i4\_real;

        g\_imag[1] <= i0\_imag - i4\_imag;

        g\_real[2] <= i2\_real + i6\_real;

        g\_imag[2] <= i2\_imag + i6\_imag;

        g\_real[3] <= i2\_real - i6\_real;

        g\_imag[3] <= i2\_imag - i6\_imag;

        g\_real[4] <= i1\_real + i5\_real;

        g\_imag[4] <= i1\_imag + i5\_imag;

        g\_real[5] <= i1\_real - i5\_real;

        g\_imag[5] <= i1\_imag - i5\_imag;

        g\_real[6] <= i3\_real + i7\_real;

        g\_imag[6] <= i3\_imag + i7\_imag;

        g\_real[7] <= i3\_real - i7\_real;

        g\_imag[7] <= i3\_imag - i7\_imag;

        stage <= 3'b01;

      end

      else if (stage > 0 ) begin

        case(stage)

          3'b001:begin

            h\_real[0] <= g\_real[0] + g\_real[2];        // h0\_real = g0\_real + g2\_real

            h\_imag[0] <= g\_imag[0] + g\_imag[2];        // h0\_imag = g0\_imag + g2\_imag

            h\_real[1] <= g\_real[1] + g\_imag[3];   // h1\_real = g1\_real - (j\*g3\_real)

            h\_imag[1] <= g\_imag[1] - g\_real[3];        // h1\_imag = g1\_imag - (j\*g3\_imag)

            h\_real[2] <= g\_real[0] - g\_real[2];        // h2\_real = g0\_real - g2\_real

            h\_imag[2] <= g\_imag[0] - g\_imag[2];        // h2\_imag = g0\_imag - g2\_imag

            h\_real[3] <= g\_real[1] - g\_imag[3];   // h3\_real = g1\_real + (j\*g3\_real)

            h\_imag[3] <= g\_imag[1] + g\_real[3];        // h3\_imag = g1\_imag + (j\*g3\_imag)

            h\_real[4] <= g\_real[4] + g\_real[6];        // h4\_real = g4\_real + g6\_real

            h\_imag[4] <= g\_imag[4] + g\_imag[6];        // h4\_imag = g4\_imag + g6\_imag

            h\_real[5] <= g\_real[5] + g\_imag[7];   // h5\_real = g5\_real - (j\*g7\_real)

            h\_imag[5] <= g\_imag[5] - g\_real[7];        // h5\_imag = g5\_imag - (j\*g7\_imag)

            h\_real[6] <= g\_real[4] - g\_real[6];        // h6\_real = g4\_real - g6\_real

            h\_imag[6] <= g\_imag[4] - g\_imag[6];        // h6\_imag = g4\_imag - g6\_imag

            h\_real[7] <= g\_real[5] - g\_imag[7];   // h7\_real = g5\_real + (j\*g7\_real)

            h\_imag[7] <= g\_imag[5] + g\_real[7];        // h7\_imag = g5\_imag + (j\*g7\_imag)

          end

          3'b010:begin

            temp0 <= h\_real[5] + h\_imag[5];

            temp1 <= h\_imag[5] - h\_real[5];

            temp2 <= h\_imag[7] - h\_real[7];

            temp3 <= (-(h\_imag[7] + h\_real[7]));

          end

          3'b011:begin

            h5\_prime\_real <= temp0 \* 16'b0000000010110100;

            h5\_prime\_imag <= temp1 \* 16'b0000000010110100;

            h7\_prime\_real <= temp2 \* 16'b0000000010110100;

            h7\_prime\_imag <= temp3 \* 16'b0000000010110100;

          end

          3'b100:begin

            i\_real[0] <= h\_real[0] + h\_real[4];  // i0\_real = h0\_real + h4\_real

            i\_imag[0] <= h\_imag[0] + h\_imag[4];  // i0\_imag = h0\_imag + h4\_imag

            i\_real[1] <= h\_real[1] + h5\_prime\_real\_16;  // i1\_real = h[1]\_real + Re((W8 ^ 1)\*h[5])

            i\_imag[1] <= h\_imag[1] + h5\_prime\_imag\_16;  // i1\_imag = h[1]\_imag + Im((W8 ^ 1)\*h[5])

            i\_real[2] <= h\_real[2] + h\_imag[6];  // i2\_real = h2\_real - (j\*h6\_real)

            i\_imag[2] <= h\_imag[2] - h\_real[6];  // i2\_imag = h2\_imag - (j\*h6\_imag)

            i\_real[3] <= h\_real[3] + h7\_prime\_real\_16;  // i3\_real = h[3]\_real + Re((W8^3)\*h[7])

            i\_imag[3] <= h\_imag[3] + h7\_prime\_imag\_16;  // i3\_imag = h[3]\_imag + Im((W8^3)\*h[7])

            i\_real[4] <= h\_real[0] - h\_real[4];  // i4\_real = h0\_real - h4\_real

            i\_imag[4] <= h\_imag[0] - h\_imag[4];  // i4\_imag = h0\_imag - h4\_imag

            i\_real[5] <= h\_real[1] - h5\_prime\_real\_16;  // i5\_real = h[1]\_real - Re((W8^1)\*h[5])

            i\_imag[5] <= h\_imag[1] - h5\_prime\_imag\_16;  // i5\_imag = h[1]\_imag - Im((W8^1)\*h[5])

            i\_real[6] <= h\_real[2] - h\_imag[6];  // i6\_real = h2\_real + (j\*h6\_real)

            i\_imag[6] <= h\_imag[2] + h\_real[6];  // i6\_imag = h2\_imag + (j\*h6\_imag)

            i\_real[7] <= h\_real[3] - h7\_prime\_real\_16;  // i7\_real = h[3]\_real -Re((W8^3)\*h[7])

            i\_imag[7] <= h\_imag[3] - h7\_prime\_imag\_16;  // i7\_imag = h[3]\_imag - Im((W8^3)\*h[7])

            ready <= 1'b1;

          end

        endcase

        if(stage == 3'b100) begin

          stage <= 2'b00;

        end

        else begin

          stage <= stage + 2'b01;

        end

      end

      else begin

        if (write) begin

          i0\_real<= in0\_real;  i0\_imag <= in0\_imag;

          i1\_real<= in1\_real;  i1\_imag <= in1\_imag;

          i2\_real<= in2\_real;  i2\_imag <= in2\_imag;

          i3\_real<= in3\_real;  i3\_imag <= in3\_imag;

          i4\_real<= in4\_real;  i4\_imag <= in4\_imag;

          i5\_real<= in5\_real;  i5\_imag <= in5\_imag;

          i6\_real<= in6\_real;  i6\_imag <= in6\_imag;

          i7\_real<= in7\_real;  i7\_imag <= in7\_imag;

        end

        ready <= 1'b0;

      end

    end

  end

always@(\*) begin

  h5\_prime\_real\_16 <= h5\_prime\_real[23:8];

  h5\_prime\_imag\_16 <= h5\_prime\_imag[23:8];

  h7\_prime\_real\_16 <= h7\_prime\_real[23:8];

  h7\_prime\_imag\_16 <= h7\_prime\_imag[23:8];

end

  assign out0\_real = i\_real[0]; assign out0\_imag = i\_imag[0];

  assign out1\_real = i\_real[1]; assign out1\_imag = i\_imag[1];

  assign out2\_real = i\_real[2]; assign out2\_imag = i\_imag[2];

  assign out3\_real = i\_real[3]; assign out3\_imag = i\_imag[3];

  assign out4\_real = i\_real[4]; assign out4\_imag = i\_imag[4];

  assign out5\_real = i\_real[5]; assign out5\_imag = i\_imag[5];

  assign out6\_real = i\_real[6]; assign out6\_imag = i\_imag[6];

  assign out7\_real = i\_real[7]; assign out7\_imag = i\_imag[7];

endmodule

**FFT testbench:**

`include "fft.v"

`timescale 1ns/1ps

module fft\_tb();

    parameter CLOCK\_PERIOD = 5; // 10 MHz clock

    reg sys\_clk, sys\_rst\_n;

    reg write, start;

    reg signed [15:0]  in0\_real  ,in0\_imag;

    reg signed [15:0]  in1\_real  ,in1\_imag;

    reg signed [15:0]  in2\_real  ,in2\_imag;

    reg signed [15:0]  in3\_real  ,in3\_imag;

    reg signed [15:0]  in4\_real  ,in4\_imag;

    reg signed [15:0]  in5\_real  ,in5\_imag;

    reg signed [15:0]  in6\_real  ,in6\_imag;

    reg signed [15:0]  in7\_real  ,in7\_imag;

    wire ready;

    wire signed [15:0] out0\_real ,out0\_imag;

    wire signed [15:0] out1\_real ,out1\_imag;

    wire signed [15:0] out2\_real ,out2\_imag;

    wire signed [15:0] out3\_real ,out3\_imag;

    wire signed [15:0] out4\_real ,out4\_imag;

    wire signed [15:0] out5\_real ,out5\_imag;

    wire signed [15:0] out6\_real ,out6\_imag;

    wire signed [15:0] out7\_real ,out7\_imag;

    fft DUT(.CLK(sys\_clk), .RST\_N(sys\_rst\_n), .write(write), .start(start), .in0\_real(in0\_real) , .in0\_imag(in0\_imag), .in1\_real(in1\_real), .in1\_imag(in1\_imag),

 .in2\_real(in2\_real)  ,.in2\_imag(in2\_imag), .in3\_real(in3\_real)  ,.in3\_imag(in3\_imag), .in4\_real(in4\_real)  ,.in4\_imag(in4\_imag), .in5\_real(in5\_real)  ,.in5\_imag(in5\_imag),

 .in6\_real(in6\_real)  ,.in6\_imag(in6\_imag), .in7\_real(in7\_real)  ,.in7\_imag(in7\_imag), .ready(ready), .out0\_real(out0\_real) ,.out0\_imag(out0\_imag), .out1\_real(out1\_real),

 .out1\_imag(out1\_imag), .out2\_real(out2\_real) ,.out2\_imag(out2\_imag), .out3\_real(out3\_real) ,.out3\_imag(out3\_imag), .out4\_real(out4\_real) ,.out4\_imag(out4\_imag),

 .out5\_real(out5\_real) ,.out5\_imag(out5\_imag), .out6\_real(out6\_real) ,.out6\_imag(out6\_imag), .out7\_real(out7\_real) ,.out7\_imag(out7\_imag));

    // VCD dump

    initial begin

        $dumpfile("fft.vcd");

        $dumpvars(0, DUT);

    end

    // System clock generator

    always begin

        #(CLOCK\_PERIOD/2) sys\_clk = ~sys\_clk;

    end

    // Initial Condition

    initial begin

        sys\_clk = 1'b0; sys\_rst\_n = 1'b0;

        write = 1'b0; start = 1'b0;

        in0\_real=16'h00\_00  ;in0\_imag = 0;

        in1\_real=16'h01\_00  ;in1\_imag = 0;

        in2\_real=16'h02\_00  ;in2\_imag = 0;

        in3\_real=16'h03\_00  ;in3\_imag = 0;

        in4\_real=16'h04\_00  ;in4\_imag = 0;

        in5\_real=16'h05\_00  ;in5\_imag = 0;

        in6\_real=16'h06\_00  ;in6\_imag = 0;

        in7\_real=16'h07\_00  ;in7\_imag = 0;

        #CLOCK\_PERIOD sys\_rst\_n = 1'b1;

        #CLOCK\_PERIOD write = 1'b1;

        #CLOCK\_PERIOD start = 1'b1;

        #CLOCK\_PERIOD start = 1'b0;

        #(CLOCK\_PERIOD/2) $display("START OF SIMULATION (Time: %g ns)", $time);

        #10000;

        $finish;

    end

endmodule

**synth.ys**

# read design

read\_verilog fft.v

hierarchy -check

hierarchy -top fft

#flatten

# high level synthesis

proc; opt; clean

fsm; opt clean

memory; opt; clean

# low level synthesis

techmap; opt; clean

# map to target architecture

dfflibmap -liberty stdcells.lib

abc -liberty stdcells.lib

# split larger signals

splitnets -ports; opt; clean

# write synthesis output

write\_verilog synth.v

write\_spice synth.sp

#print synthesis reports

stat

stat -liberty stdcells.lib

show -format svg -prefix /home/ujjwal/Desktop/ESDCS/ESDCS\_project/syme fft

**readme file:**

**# ESDCS\_FFT\_Project**

Project for the partial grading of Course: Efficient and Secure Digital Systems

**## Project Objectives**

1. Design a digital circuit for Fast Fourier Transform (FFT) in Verilog

2. Create an appropriate test bench in Verilog to test the design

3. Simulate the design using Icarus Verilog and visualize the simulated VCD waveforms using GTKWave to verify functionality

4. Synthesize the design using Yosys with the Nangate Open Cell 45nm standard cell library in the typical TT corner

5. Analyze timing and power of the implementation using OpenSTA

**## Project Statement**

- Input 𝑥 = {𝑥0, 𝑥1, ⋯ , 𝑥7}

- Output 𝑋 = {𝑋0, 𝑋1, ⋯ , 𝑋7}

- Output 𝑋 is the 8-point FFT of input x

- Each input element 𝑥𝑚 (for 𝑚 = 0,1, … , 7 ) is a complex number with real and imaginary parts each represented as a signed 16-bit fixed point quantity with 1 sign bit, 7 bits for decimal part and 8 bits for fractional part

- Each output element 𝑋𝑘 (for 𝑘 = 0,1, … , 7 ) is a complex number with real and imaginary parts each represented as a signed 16-bit fixed-point quantity with 1 sign bit, 7 bits for decimal part and 8 bits for fractional part

- Additional Signals

    - |CLK |Input| Clock| signal|

      |----|-----|------|-------|

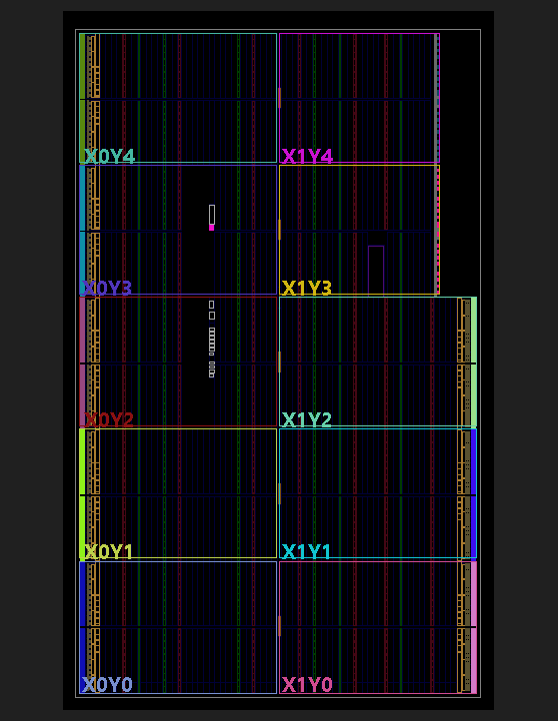
      | RST\_N | Input | Active-low | reset signal |

      | write | Input | Input | write signal |

      | start | Input | FFT computation | start signal |

      | ready | Output | FFT computation | done signal  |

**Synthesized Circuit:**



## References

1. Wikipedia – Fast Fourier Transform
2. NPTEL – Hardware Modelling using Verilog by Prof. Indranil Sen Gupta
3. “The Fast Fourier Transform (FFT): Most Ingenious Algorithm Ever?” Youtube.com by

Reducible

1. “DSP Tricks: Computing inverse FFTs using the forward FFT” – embedded.com