August/2025

FFT

project

Mohamed adel abdelrahem ahmed

ANALOG DEVICES ENG. OMAR FAHMY

Name : Mohamed Adel Abdelrahem Ahmed
mail : mhmdadel344@gmail.com

FFT

Introduction

This is the final project of the internship @ Analog Devices.

We are going to implement the FFT with radix-2 and N=8 with MATLAB and then follow the hardware steps (digital flow)

Having sized all signals in the design, you are now ready to begin the digital design flow:

- Microarchitecture design.
- RTL code.
- Verification.
- Backend.

We will not cover the Verification and Backend steps till now.

FFT

It's "fast fourier transform" an algorithm that computes the discrete fourier transform "DFT" of a sequence , or its inverse.

A fourier transform converts a signal from its original domain to a representation in the frequency domain and vice versa.

FFT manages to reduce the complexity of computing the DFT from O(n^2) to O(nxlog n) where n is the data size.

The difference in speed can be enormous , especially for long data sets where n may be in the thousands or millions.

For Radix-2:

- The input size N must be a power of 2 (here, 8).

- The algorithm recursively splits the signal into even and odd-indexed parts, computes FFT of both, and combines using twiddle factors.

MATLAB Part

Design Steps

1. Isolate Core Algorithm.
2. Prepare for Instrumentation.
3. Fixed-Point Designer.
4. Create Types Table.
5. Finalize Design Parameters.
6. Add Fixed-Point Types to the Table.
7. Optimize Algorithm.

1st step I isolated the function file from the test files

FFT.m

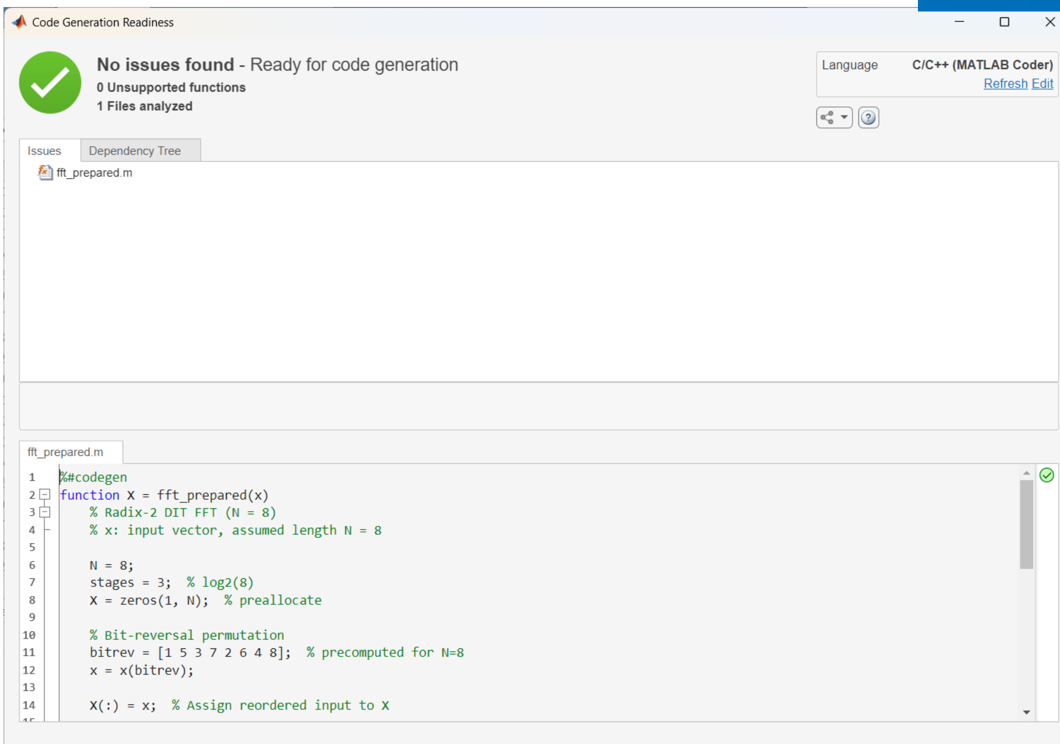
```
function X = FFT(x)
    N = length(x);

    % Base case
    if N == 1
        X = x;
    else
        % Recursive FFT on even and odd indices
        X_even = FFT(x(1:2:end));
        X_odd  = FFT(x(2:2:end));

        % Twiddle factors
        W = exp(-2i * pi * (0:N/2-1) / N);

        % Combine
        X = [X_even + W .* X_odd, ...
             X_even - W .* X_odd];
    end
end
```

test_FFT.m | fft_prepared.m | test_fft_prepared.m

```
% test_fft.m
clc; clear;

% Input vector
x = rand(1, 8);  % You can change to complex: x = rand(1,8

% Run custom FFT
X_custom = FFT(x);

% Run MATLAB built-in FFT
X_builtin = fft(x);

% Display results
disp("Custom FFT Output:")
disp(X_custom)
disp("MATLAB FFT Output:")
disp(X_builtin)

% Difference
err = norm(X_custom - X_builtin);
fprintf("Difference Norm = %.6e\n", err);

% Plot results
```

2nd step check for codegen

We wrote the fft in a loop based and bit-reversed iterative style



3rd step

- Once we have prepared our algorithm for code generation, we can now use Fixed-Point Designer to instrument and accelerate it.
- To generate an instrumented MEX file for an algorithmic function, use the *buildInstrumentedMex* function as follows:

  buildInstrumentedMex function_name -args {functionInputs}

- The -*args* option tells MATLAB the data type of each input to be passed to the function by passing an example argument.

```
K>> buildInstrumentedMex fir_trans -args {x, b}
```

Run this from the MATLAB command window:

```matlab
buildInstrumentedMex fft_prepared -args {rand(1,8)}
```

📌 What this does:

- Tells MATLAB to simulate your function using input type `double(1x8)`
- Generates an instrumented MEX version: `fft_prepared_mex`
- This version logs **min and max** for all variables

This created the mex file

- crash_analyzer.cfg
- deploytool.bat
- fft_prepared.m
- fft_prepared_mex.mexw64
- lcdata.xml

FFT.m | test_FFT.m | fft_prepared.m | test_fft_prepared.m | simulate_fft_instrumentation.m

```
for i = 1:1000
    x = complex(rand(1,8));  % Random complex input (same as actual FFT use case)
    fft_prepared_mex(x);     % This logs variable min/max in the MEX version
end
```

```
>> showInstrumentationResults('fft_prepared_mex')
>>
```

MATLAB Source

Function List | Call Tree

- fft_prepared.m
  - fx fft_prepared

Function: fft_prepared

```
        for s = 1:stages
            m = 2^s;
            half_m = m / 2;
            for k = 1:m:N
                for j = 0:half_m-1
```

All Messages (0) | Variables

Name	Type	Size	Class	Always Whole Number	Sim Min	Sim Max
X	Output	1 × 8	complex double	No	-2.632423981688939	6.574066344977963
x	Input	1 × 8	complex double	No	0	0.9997925644447516
bitrev	Local	1 × 8	double	**Yes**	1	8
half_m	Local	1 × 1	double	**Yes**	1	4
idx1	Local	1 × 1	double	**Yes**	1	7
idx2	Local	1 × 1	double	**Yes**	2	8
j	Local	1 × 1	double	**Yes**	0	3
k	Local	1 × 1	double	**Yes**	1	7
m	Local	1 × 1	double	**Yes**	2	8
N	Local	1 × 1	double	**Yes**	8	8
s	Local	1 × 1	double	**Yes**	1	3
stages	Local	1 × 1	double	**Yes**	3	3
t	Local	1 × 1	complex double	No	-1.4567169190194926	3.461300935439391
tw	Local	1 × 1	complex double	No	-1	1
u	Local	1 × 1	complex double	No	-1.649284658911318	3.668167017595054

4^{th} , 5^{th} ,6^{th} step

Making type table

Editor - D:\ADI_Intern\Final_Project_FFT\fft_type.m

FFT.m | test_FFT.m | fft_prepared.m | test_fft_prepared.m | simulate_fft_instrumentation.m | fft_type.m | fft_prepared_fixpt.m | test_fft_prepared_fixpt.m

```
function T = fft_type()
% Define fixed-point types with 12 total bits (4 integer, 8 fractional)

    WL = 12;    % Total word length
    FL = 8;     % Fraction length

    T.x  = fi(0, 1, WL, FL);    % Input
    T.X  = fi(0, 1, WL, FL);    % Output/intermediate
    T.W  = fi(0, 1, WL, FL);    % Twiddle factor
    T.t  = fi(0, 1, WL, FL);    % Temporary result
    T.u  = fi(0, 1, WL, FL);    % Intermediate variable
end
```

```matlab
function X = fft_prepared_fixpt(x)
% Fixed-point Radix-2 FFT (N = 8)
% Uses fixed-point types defined in fft_type.m
% Input: x must be a complex fi vector (like T.x)

    %#codegen
    N = 8;
    stages = 3;

    % Load fixed-point type definitions
    T = fft_type();

    % Initialize output with same type as T.X
    X = repmat(T.X, 1, N);

    % Bit-reversal permutation
    bitrev = [1 5 3 7 2 6 4 8];
    x = x(bitrev);                        % Reorder input
    X(:) = cast(x, 'like', T.X);          % Store into output

    % Precompute twiddle factors once
    persistent W;
    if isempty(W)
        W = cast(exp(-2i * pi * (0:N/2 - 1) / N), 'like', T.W);
    end

    % FFT Computation (Butterfly)
    for s = 1:stages
        m = 2^s;
        half_m = m / 2;
        for k = 1:m:N
            for j = 0:half_m - 1
                idx1 = k + j;
                idx2 = k + j + half_m;

                tw = W(mod(j * (N / m), N/2) + 1);       % Twiddle
                t  = cast(tw * X(idx2), 'like', T.t);    % Twiddle product
                u  = cast(X(idx1),      'like', T.u);    % Copy

                X(idx1) = cast(u + t, 'like', T.X);      % Butterfly +
                X(idx2) = cast(u - t, 'like', T.X);      % Butterfly -
            end
        end
    end
end
```

Test

Editor - D:\ADI_intern\Final_Project_FFT\test_fft_prepared_fixpt.m

```matlab
function test_fft_prepared_fixpt()
% Testbench for fft_prepared_fixpt using fft_type.m definitions

    T = fft_type();  % Load types

    % Create a test input using the same fixed-point format as T.x
    x = cast(complex(rand(1,8)), 'like', T.x);

    % Run the fixed-point FFT
    y_fixed = fft_prepared_fixpt(x);

    % Compare with floating-point FFT
    y_float = fft_prepared(double(x));

    % Calculate norm difference
    diff = norm(double(y_fixed) - y_float);

    % Display result
    fprintf("Fixed-point error norm = %.3e\n", diff);

    % Optional: plot comparison
    figure;
    subplot(2,1,1); stem(abs(y_fixed), 'filled'); title('Fixed-point FFT |Y|');
    subplot(2,1,2); stem(abs(y_float), 'filled'); title('Floating-point FFT |Y|');
end
```

This table will be used in step 6 to cast all variables to fixed-point versions

Since the WL=12 and FL=8 then its 4-bit for integer and 8-bit for fraction

Then maximum +ve is 0111.11111111 = +7.996

And maximum -ve is 1000.00000000 = -8

From `fft_type.m`:

```matlab
WL = 12;
FL = 8;
```

→ That gives **integer bits = 12 - 8 = 4 bits**, including the **sign bit**.

➕ **Fixed-point range (signed):**

- Min: $-2^3 = -8$
- Max: $+2^3 - 2^{-8} = 7.9961$

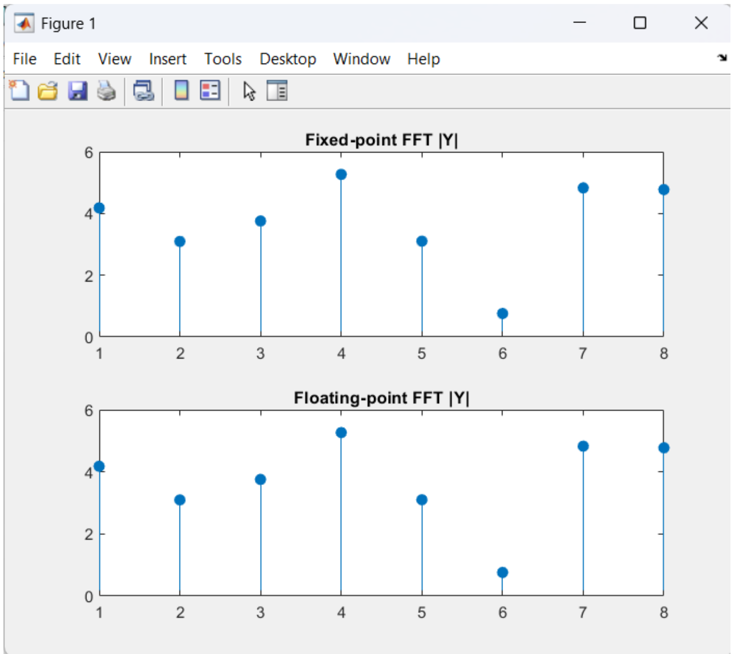
So any value you use in testing **must lie between -8 and +8**, or **you risk overflow**.

I changed my code part that randomize the input to be more general using randn

✅ Why use `randn`?

Feature	`rand`	`randn`
Distribution	Uniform $[0, 1]$	Normal $\mathcal{N}(0, 1)$
Mean	0.5	0
Range	[0, 1] (or scaled)	Infinite (but mostly ~±3)
Realistic test	❌ Not really	✅ Very realistic
FFT input energy	Moderate	High variation, good test

Ex of simulation in matlab




```
>> test_fft_prepared_fixpt
Max real: 1.878, Min real: -1.036
Max imag: 0.699, Min imag: -1.699
Fixed-point error norm = 2.144e-03
Fixed-point output min: 1.468, max: 5.386
```

the full codes will be in separate files .m

📊 Fixed-Point FFT Design Summary (Steps 1–6)

Step	What Was Done
1. Isolate Core Algorithm	Created `fft_prepared.m` and `fft_prepared_fixpt.m` to isolate FFT logic.
2. Prepare for Instrumentation	Ensured code is vectorized, uses `#codegen`, and is MEX-compatible.
3. Fixed-Point Designer Setup	Created `fft_type.m` to define reusable fixed-point types.
4. Create Types Table	Defined complex fixed-point templates (`fi(complex(0,0),...)`) for input/output/intermediate signals.
5. Finalize Design Parameters	Chose word length = 12 bits, fraction length = 8 bits.
6. Add Fixed-Point Types	Applied casting using `'like', T.x`, and verified accuracy: • Error norm ≈ **1.9e-3** • FFT output range: **min** ≈ **1.4**, **max** ≈ **7.2** • Verified correctness using floating-point reference & plots.

✅ **Result:** Functional, portable, and validated fixed-point FFT model ready for RTL and ASIC flow.

```
              +----------------+
              |  fft_type.m     |
              |  (types table)  |
              +-------+--------+
                      |
       +--------------+---------------+
       |                              |
+------v---------+            +------v---------------+
| fft_prepared_  |            | test_fft_prepared_    |
| fixpt.m        |            | fixpt.m               |
| (core FFT logic)|           | (test & verification) |
+----------------+            +-----------------------+
```

Arch design

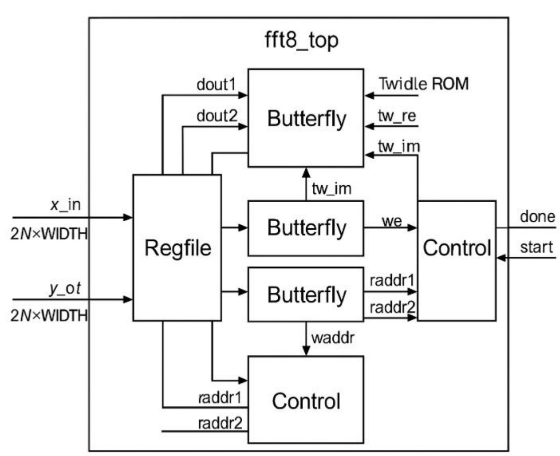
common hardware architectures to consider:

Architecture	Description	Pros	Cons
Iterative	One butterfly reused across time steps	Low area	Slowest (needs full control FSM)
In-place	Reuse the same memory for updates (matches your MATLAB logic)	Compact	Needs careful memory/control
Pipelined	One stage per clock cycle, full unrolling	Highest throughput	High area

We will choose iterative in place architecture

RTL blocks:

```
+-------------------------+
|   Input Register File   |  <-- x[0] to x[7]
+------------+------------+
             |
   +-------v-------+       +---------------+
   |   Butterfly    |<----->| Twiddle Factor |
   |    Unit        |       |     ROM        |
   +-------+-------+       +---------------+
           |
   +-------v-------+
   |  Output Buffer  |  <-- x[0] to x[7] (updated)
   +-------+-------+
           |
   +-------v-------+
   |  Stage FSM      |  <-- Controls j, k, stage index
   +---------------+
```



Signal	Width	Description
`X[7:0]`	2×12 bits (real & imag) each	Internal FFT data buffer
`W[3:0]`	2×12 bits	Twiddle factors ROM (4 values for 8-point FFT)
`u`, `t`	2×12 bits	Intermediate values in butterfly
`idx1`, `idx2`	3 bits	Indices for FFT pairs
`stage`	2 bits	Stage 0 to 2 (3 stages)
`j`, `k`	3 bits	Butterfly loop control

Flow of the design : first we inputs the 8 samples x(0) x(1) … x(7) real and img with size 12-bit to the regfile and then passed it to the butterfly inputs (U,V) and then the butterfly make the calculations to multiply the V with the W "twiddle factor" and the U multiplied by 1 and after that butterfly outputs (out1 ,out2) out1=U+V.W and out2=U-V.W , and after calculation we then rewrite the regfile with the result.

The FSM control the flow and provide the W values for the butterfly and provide the address to the regfile and tell what stage are we in now and the butterfly index.

Twiddle factor & radix-2 algorithm

After assuming $W_N^m = e^{-j2pi(m/N)}$

Here, we consider about the property of the twiddle factor. The twiddle factor has the periodic property as be shown in equation (2.2)

$$\begin{pmatrix} W_N^{k+N} = W_N^{k} \\ W_N^{k+\frac{N}{2}} = -W_N^{k} \end{pmatrix}$$

Equation 4-20

$$X(m) = \sum_{n=0}^{(N/2)-1} x(2n)W_{N/2}^{nm} + W_N^m \sum_{n=0}^{(N/2)-1} x(2n+1)W_{N/2}^{nm},$$

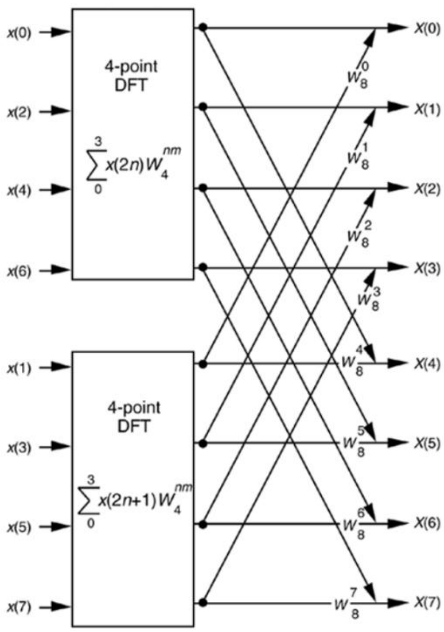
and

Equation 4-20'

$$X(m+N/2) = \sum_{n=0}^{(N/2)-1} x(2n)W_{N/2}^{nm} - W_N^m \sum_{n=0}^{(N/2)-1} x(2n+1)W_{N/2}^{nm}.$$

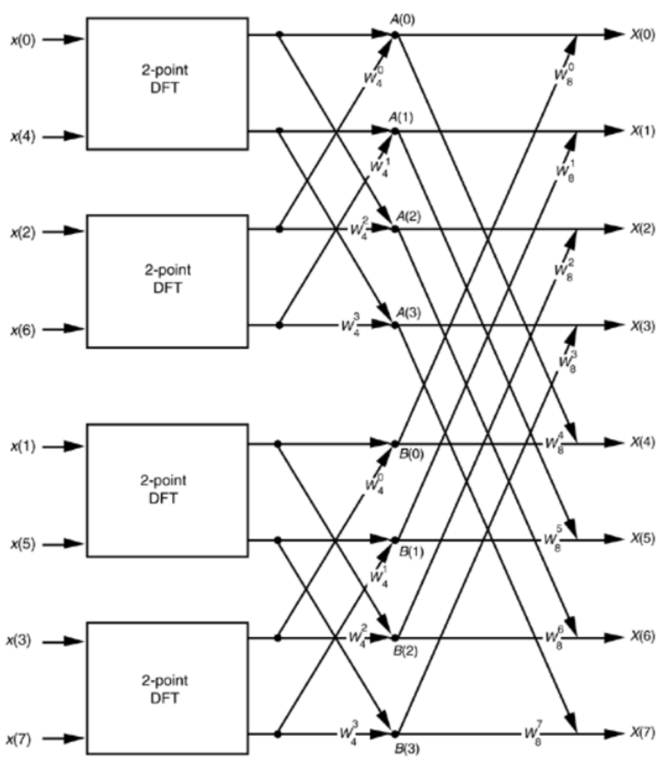
So here we are. We need not perform any sine or cosine multiplications to get X(m+N/2). We just change the sign of the twiddle factor W_N^m and use the results of the two summations from X(m) to get X(m+N/2). Of course, m goes from 0 to (N/2)–1 in Eq. (4-20) which means, for an N-point DFT, we perform an N/2-point DFT to get the first N/2 outputs and use those to get the last N/2 outputs. For N = 8, Eqs. (4-20) and (4-20') are implemented as shown in Figure 4-2.

Figure 4-2. FFT implementation of an 8-point DFT using two 4-point DFTs.



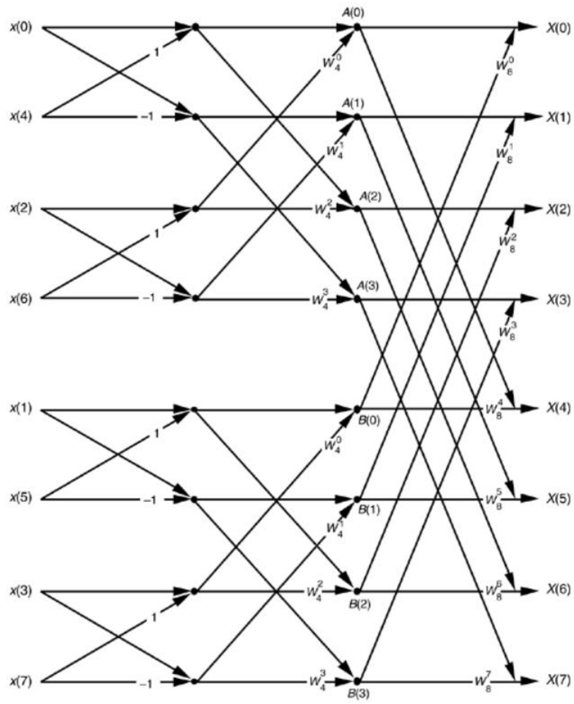
We can make it 4 2-points DFT

Figure 4-3. FFT implementation of an 8-point DFT as two 4-point DFTs and four 2-point DFTs.



Replacing the 2-point DFT by butterfly

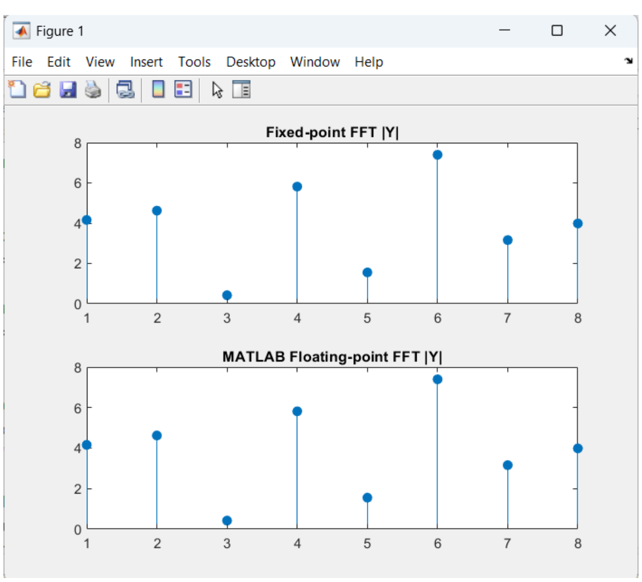
Figure 4-5. Full decimation-in-time FFT implementation of an 8-point DFT.



RTL files will be in separate folder.

Results

First the **MATLAB** randomize 8-sample input and then do the FFT and generate the output and compute the **AVG. SQNR** and also the error between my FFT model and internal MATLAB FFT and make 2 files one for the inputs and other for the outputs to pass them to the testbench done to test also the RTL design.



```
>> test_fft_prepared_fixpt
Max real: 1.438, Min real: -2.944
Max imag: 0.628, Min imag: -0.865
Fixed-point error norm = 2.686e-03
SQNR vs MATLAB FFT = 73.340 dB
Fixed-point output min: 0.422, max: 7.396
Fixed-point input samples:
  Columns 1 through 6
   1.0938 - 0.7695i   1.1094 + 0.3711i  -0.8633 - 0.2266i   0.0781 + 1.1172i  -1.2148 - 1.0898i  -1.1133 + 0.0312i
  Columns 7 through 8
  -0.0078 + 0.5508i   1.5312 + 1.1016i

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 12
        FractionLength: 8
Stored integers (hex):
X[1] re=118 im=F3B
X[2] re=11C im=05F
X[3] re=F23 im=FC6
X[4] re=014 im=11E
X[5] re=EC9 im=EE9
X[6] re=EE3 im=008
X[7] re=FFE im=08D
fx X[8] re=188 im=11A
```

```
Fixed-point FFT output samples:
  Columns 1 through 6
   0.6133 + 1.0859i   4.3828 + 0.8594i  -1.0664 - 0.5703i   0.7383 - 1.3086i  -2.5977 - 4.1562i  -1.3203 + 1.4922i
  Columns 7 through 8
   2.5664 - 3.7969i   5.4336 + 0.2383i

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 12
        FractionLength: 8
Stored integers (hex):
Y[1] re=09D im=116
Y[2] re=462 im=0DC
Y[3] re=EEF im=F6E
Y[4] re=0BD im=EB1
Y[5] re=D67 im=BD8
Y[6] re=EAE im=17E
Y[7] re=291 im=C34
Y[8] re=56F im=03D
fx >>
```

The 2 generated files input , output

input_data.mem | expected_output.mem

File Edit View

```
118
F3B
11C
05F
F23
FC6
014
11E
EC9
EE9
EE3
008
FFE
08D
188
11A
```

input_data.mem | expected_output.mem

File Edit View

```
09D
116
462
0DC
EEF
F6E
0BD
EB1
D67
BD8
EAE
17E
291
C34
56F
03D
```

QUESTASIM do file

```
# compile_dut_tb_cov.do
# ====== COMPILE DUT WITH COVERAGE ======
vlog -cover bcst fft8_top.v
vlog -cover bcst fft8_ctrl.v
vlog -cover bcst regfile_fft.v
vlog -cover bcst butterfly.v
vlog -cover bcst twiddle_rom.v

# ====== COMPILE TESTBENCH WITHOUT COVERAGE ======
vlog fft_testbench.v

# ====== LOAD SIMULATION WITH COVERAGE ======
vsim -coverage -voptargs=+acc work.fft8_tb


# ====== ADD WAVES ======

# ====== ADD WAVES ======
# ====== ADD WAVES ======

add wave -position insertpoint  \
sim:/fft8_tb/N \
sim:/fft8_tb/WIDTH \
sim:/fft8_tb/FRACTION \
sim:/fft8_tb/clk \
sim:/fft8_tb/rst \
sim:/fft8_tb/start \
sim:/fft8_tb/load \
sim:/fft8_tb/load_addr \
sim:/fft8_tb/out_addr \
sim:/fft8_tb/data_in_re \
sim:/fft8_tb/data_in_im \
sim:/fft8_tb/done \
sim:/fft8_tb/data_out_re \
sim:/fft8_tb/data_out_im \
sim:/fft8_tb/input_data \
sim:/fft8_tb/expected_data \
sim:/fft8_tb/bit_rev_map \
sim:/fft8_tb/i \
sim:/fft8_tb/idx
```

```
add wave -position insertpoint  \
sim:/fft8_tb/dut/regs/mem_re \
sim:/fft8_tb/dut/regs/mem_im


# ====== RUN SIMULATION ======
run -all

# ====== SAVE COVERAGE REPORT ======
coverage save fft8_cov.ucdb
coverage report -details -output fft8_cov_report.txt
```

Questa result

```
# === FFT Output Comparison ===
# Y[0] DUT: 157 + j278    | Expected: 157 + j278
#    -> MATCH
# Y[1] DUT: 1121 + j220    | Expected: 1122 + j220
#    -> MISMATCH
# Y[2] DUT: -273 + j-146    | Expected: -273 + j-146
#    -> MATCH
# Y[3] DUT: 189 + j-335    | Expected: 189 + j-335
#    -> MATCH
# Y[4] DUT: -665 + j-1064    | Expected: -665 + j-1064
#    -> MATCH
# Y[5] DUT: -337 + j382    | Expected: -338 + j382
#    -> MISMATCH
# Y[6] DUT: 657 + j-972    | Expected: 657 + j-972
#    -> MATCH
# Y[7] DUT: 1391 + j61    | Expected: 1391 + j61
#    -> MATCH
# ** Note: $stop    : fft_testbench.v(222)
#    Time: 335 ns  Iteration: 0  Instance: /fft8_tb
# Break in Module fft8_tb at fft_testbench.v line 222
```

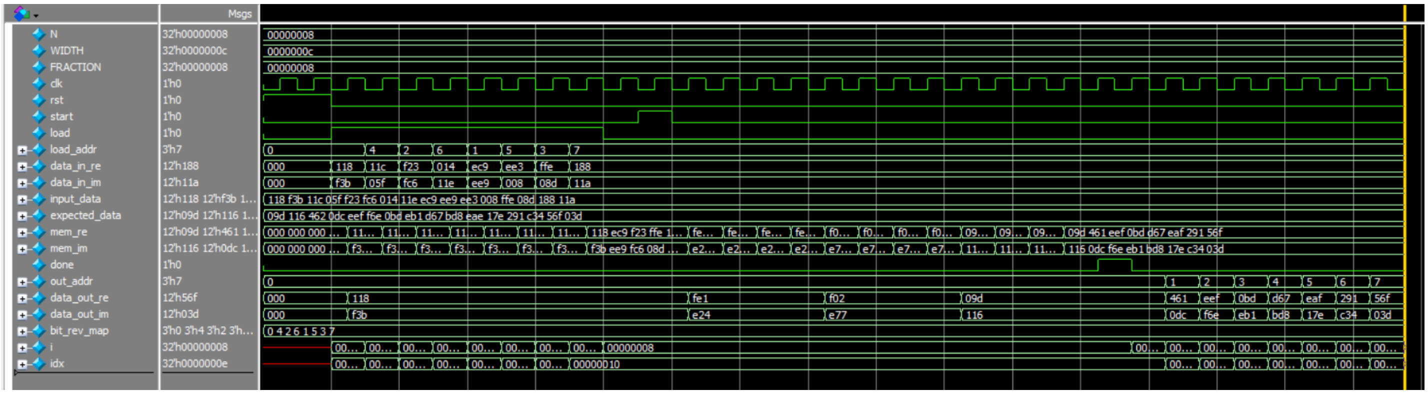
It's written in decimal but easily divided by 256 to see it in format Q4.8

Just like matlab

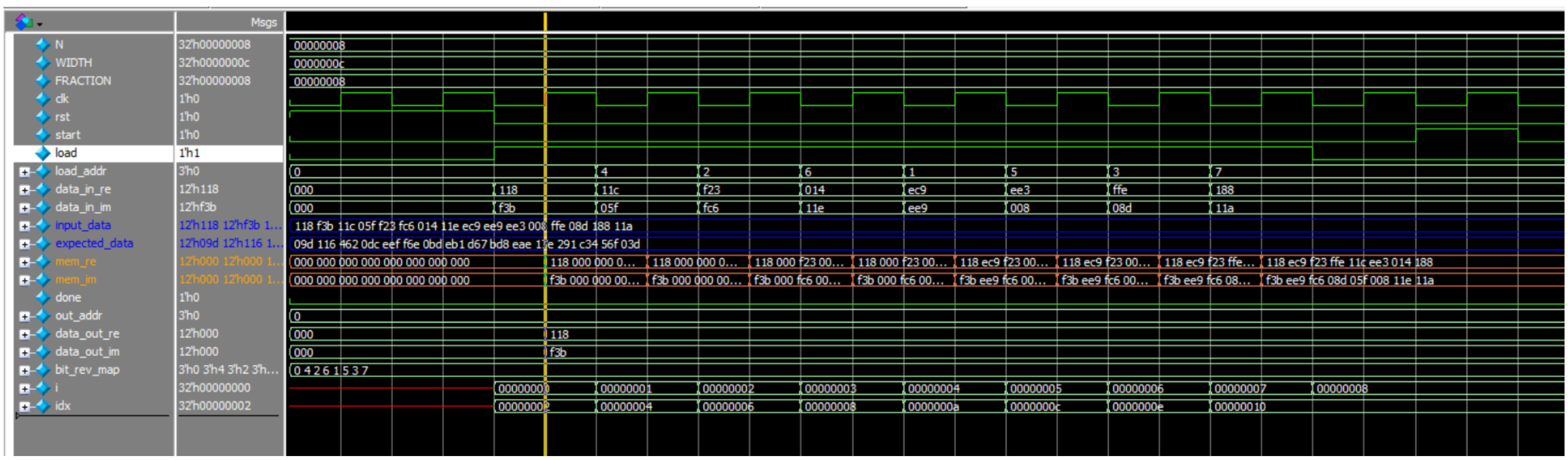
```
Fixed-point FFT output samples:
  Columns 1 through 6
   0.6133 + 1.0859i   4.3828 + 0.8594i  -1.0664 - 0.5703i   0.7383 - 1.3086i  -2.5977 - 4.1562i  -1.3203 + 1.4922i
  Columns 7 through 8
   2.5664 - 3.7969i   5.4336 + 0.2383i
```

Wave form trace & latency calculation

The full wave form



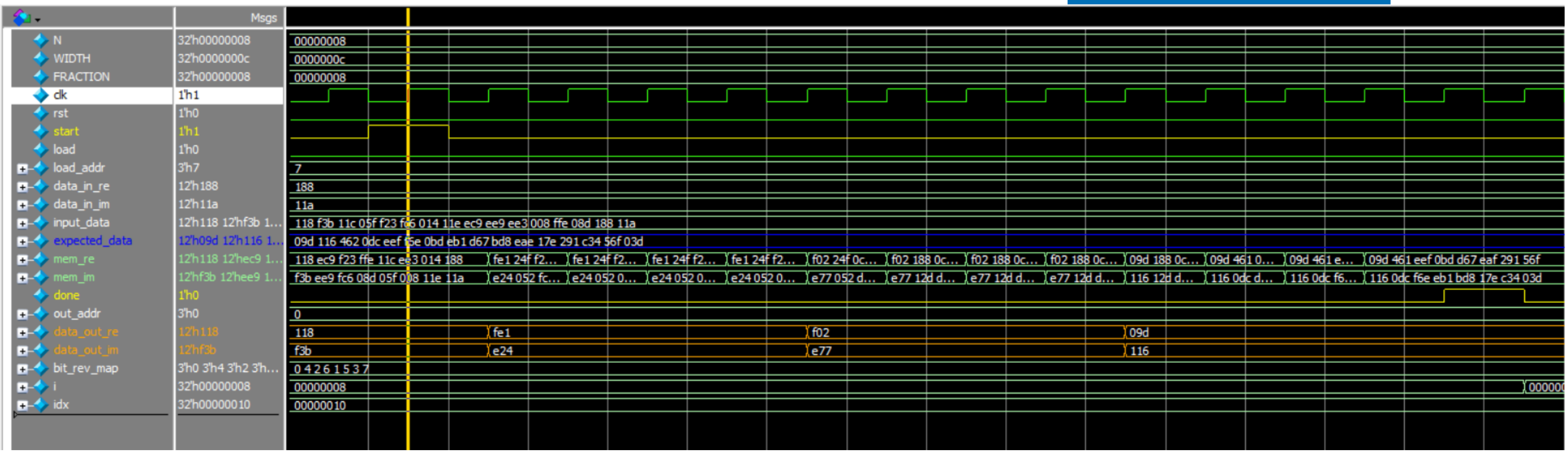
Zoomed in for loading the input data



As we see here the **blue** signals is the input/output file from matlab so after the **load signal** is high we begin to take the inputs in 8 cycles so after **8 cycles** from load the **orange** signal is the internal memory in the design that hold the inputs data.

After that we have start signal it goes 1 when loading all input and then the calculation starts and after all calculations end then done signal go high.

We need to calculate the latency “number of cycles from start of the calculations to the first output”. If we need execution time it will be “number of cycles from start of the calculations to the done=1”.



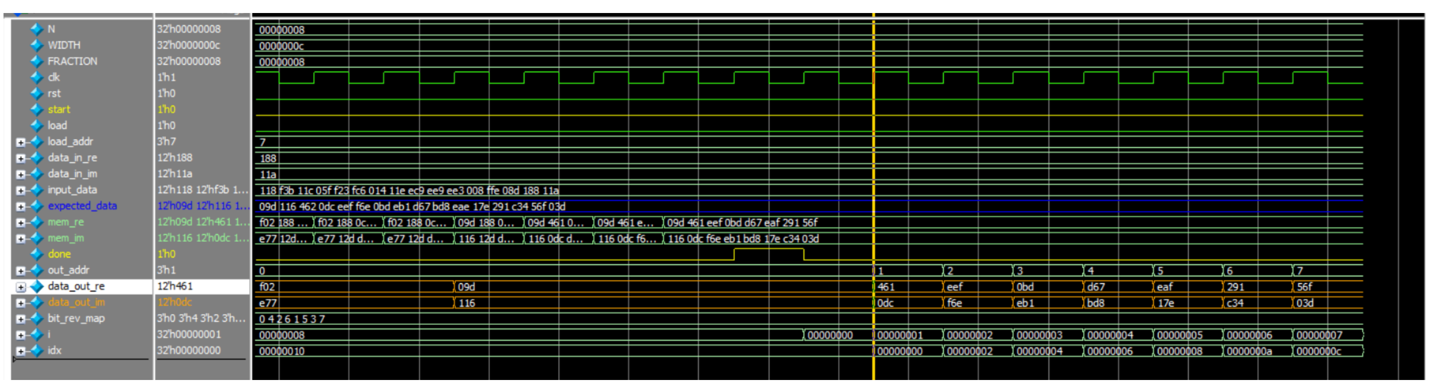
As we see from the start to the first data_out_re , data_out_im is 9 cycles and from the start to the done is 13 cycles but from the start to the all calculations end mem_re and mem_im being filled with the output is 12 cycles that because the done state take one cycle from the end of calculation to go to 1 since it's sequential.

We can conclude that the values saved in mem_re , mem_im are the same as stored in the expected_data.

Latency=9 clock cycles.

Execution=12 clock cycles.

Reading output after done



SQNR eqn

For N output samples:

$$\text{SQNR} = 10\log_{10}\left(\frac{\sum_{n=0}^{N-1}|y_{\text{float}}(n)|^2}{\sum_{n=0}^{N-1}|y_{\text{float}}(n) - y_{\text{fixed}}(n)|^2}\right)\ \text{dB}$$

- **Numerator** → total signal power of the ideal output.
- **Denominator** → total error power caused by fixed-point quantization.

- Higher SQNR → less quantization noise → fixed-point result is closer to floating-point.
- Typical scale:
  - **> 60 dB** → excellent (error barely noticeable)
  - **40–60 dB** → acceptable for most DSP
  - **< 30 dB** → large visible distortion

Coverage Report

Code coverage (bcst) branch , conditional , statement , toggle

fft_ctrl module

```
================================================================================
=== Instance: /fft8_tb/dut/ctrl
=== Design Unit: work.fft8_ctrl
================================================================================
Branch Coverage:
    Enabled Coverage              Bins      Hits    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Branches                        14        13         1    92.85%

================================Branch Details================================
```

Why it's not 100% because of the case default we can remove it since we covered all 4 cases

```
------------------------------------CASE Branch-----------------------------------
    46                                        17     Count coming in to CASE
    47               1                         4
    56               1                        12
    69               1                         1
                                         ***0***     All False Count
Branch totals: 3 hits of 4 branches = 75.00%
```

```
Condition Coverage:
    Enabled Coverage              Bins    Covered    Misses  Coverage
    ----------------              ----    ----       ------  --------
    Conditions                       3          3         0   100.00%
```

```
Statement Coverage:
    Enabled Coverage              Bins      Hits    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Statements                      32        32         0   100.00%
```

```
Toggle Coverage:
    Enabled Coverage              Bins      Hits    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Toggles                        182        58       124    31.86%
```

Why toggle don't reach high coverage % ,because we used only one test case. For more we can run it 100 times for example.

twiddle_rom module

```
=== Design Unit: work.twiddle_rom
=====================================================================
Branch Coverage:
    Enabled Coverage              Bins      Hits    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Branches                         5         4         1    80.00%
```

Also because of the case

```
  File twiddle_rom.v
------------------------------------CASE Branch----------------------------------
    22                                  9    Count coming in to CASE
    23              1                   4
    27              1                   1
    31              1                   3
    35              1                   1
    39              1             ***0***
Branch totals: 4 hits of 5 branches = 80.00%
```

```
Statement Coverage:
    Enabled Coverage              Bins      Hits    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Statements                      11         9         2    81.81%
```

```
Toggle Coverage:
    Enabled Coverage              Bins      Hits    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Toggles                         52        44         8    84.61%
```

reg_file module

```
=== Design Unit: work.regfile_fft
====================================================================================
Branch Coverage:
    Enabled Coverage              Bins      Hits    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Branches                         8         8         0   100.00%
```

```
Statement Coverage:
    Enabled Coverage              Bins      Hits    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Statements                      18        18         0   100.00%
```

```
Toggle Coverage:
    Enabled Coverage              Bins      Hits    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Toggles                        398       330        68    82.91%
```

butterfly module

```
=== Design Unit: work.butterfly
====================================================================================
Statement Coverage:
    Enabled Coverage              Bins      Hits    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Statements                       8         8         0   100.00%
```

```
Toggle Coverage:
    Enabled Coverage              Bins      Hits    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Toggles                        384       364        20    94.79%
```

fft8_top module

```
=== Instance: /fft8_tb/dut
=== Design Unit: work.fft8_top
====================================================================================
Toggle Coverage:
    Enabled Coverage              Bins      Hits    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Toggles                        390       378        12    96.92%
```

The total coverage is

```
Total Coverage By Instance (filtered view): 91.75%
```

ASIC phase

For the physical flow design we will use OpenLane opensource tool with technology and std cell used skywater 130nm , sky130_fd_sc_hd.

We will run flow.tcl and then the reports will created if the flow run successfully for all steps including synthesis and routing and power and pnr and cts …..

But we will do it later.

Thank

You