Digital VLSI Design

Final Project - Report

256 POINT FFT CORE FOR IMAGE PROCESSING:

THEORY:

Discrete Time Fourier Transform:

Like continuous time signal Fourier transform, discrete time Fourier Transform (DTFT) can be used to represent a discrete sequence into its equivalent frequency domain representation and LTI discrete time system. It is given by the following equation:

$$X(\omega) = \sum_{n=-\infty}^{\infty} x(n)e^{-j\omega n}.$$

Obtaining DFT from DTFT:

DFT is obtained from DTFT by sampling the spectrum X(ω) in frequency.
The corresponding equations are shown below:

$$X(k) = X(k\Delta\omega), \quad \Delta\omega = \frac{2\pi}{N}$$

$$X(k) = \sum_{n=0}^{N-1} x(n)e^{-j2\pi\frac{kn}{N}}$$

The above equation represents the DFT of a discrete sequence.
The following equations describe a fourier transform pair. The first equation where DFT is obtained from the discrete sequence is called as analysis equation, whereas the second equation where the sequence is obtained back from the DFT is called as synthesis equation.

$$X(k) = \sum_{n=0}^{N-1} x(n)e^{-j2\pi\frac{kn}{N}} \quad \text{analysis}$$

$$x(n) = \frac{1}{N}\sum_{k=0}^{N-1} X(k)e^{j2\pi\frac{kn}{N}} \quad \text{synthesis.}$$

Here the terms $W = e^{-j\frac{2\pi}{N}}$ are called twiddle factors and the alternative form of equations involving the twiddle factors are shown as follows:

$$X(k) = \sum_{n=0}^{N-1} x(n)W^{kn}$$

$$x(n) = \frac{1}{N}\sum_{k=0}^{N-1} X(k)W^{-kn}.$$

Computational Complexity in DFT:
Each DFT coefficient requires N complex multiplications and (N-1) complex additions
All N DFT coefficients require N^2 complex multiplications and N(N-1) complex additions.
Example: For the case of N = 64, complexity of the order of $(64)^2$ which is as high as 4096. This value increases further as the value of N increases.
So we are in need of an efficient method of performing DFT.

Fast Fourier Transform:
Fast Fourier Transform or FFT is an efficient way of performing DFT. Unlike CTFT (Continuous Time Fourier Transform), DTFT or DFT, fast fourier transform is not a transform. It is just an algorithm to compute DFT.
FFT requires a lesser number of computations to be done to obtain the DFT of a sequence.
So much of the modern technology that we have today like wireless communication, GPS and anything related to the vast field of Signal Processing relies on the insights of FFT.
These were first developed by Cooley and Tukey though the initial contributions were made as early as 1805.
The Cooley-Tukey algorithm is the most commonly used algorithm to find FFT of a given sequence and is a good example for divide and conquer.
DFT can be computed using FFT algorithm only if the number of points is a power of 2, for example N = 8,32,128,256 etc
The Fast Fourier Transform reduces the computations needed for a sequence of size N from $O(N^2)$ to $O(N\log_2 N)$. The following table shows the complexities for DFT and FFT for different lengths of input sequences (for different values of N)

N	1000	10^6	10^9
N^2	10^6	10^{12}	10^{18}
$Nlog_2N$	10^4	$20\,x\,10^6$	$30\,x\,10^9$

From the above table it is clearly evident that the computational complexity reduces by a great extent when FFT is used as opposed to when DFT is computed normally. FFT does so by exploiting the symmetries of twiddle factors, W_N terms. It is based on the principle of decomposing the computation of sequence of length N into successively smaller DFTs.

Complex conjugate symmetry

$$W_N^{k(N-n)} = W_N^{-kn} = (W_N^{-kn})^*$$

Periodicity in n,k

$$W_N^{kn} = W_N^{k(N+n)} = W_N^{(k+N)n}$$

Using the FFT algorithm greatly increases the speed of computation of DFT.
Computational Complexity:
The number of complex multiplications required to compute the N-point DFT using radix-2 FFT are $N\log_2 N$ as compared to N^2 in normal computation of DFT.
For example, let us consider the example of a case where N = 64
Here the number of complex multiplications required during normal computation of DFT are $N^2 = (64)^2 = 4096$ whereas FFT requires only 192 operations!! Speed has improved by a factor of 4096/192 = 21.33

Types of FFT algorithms:
There are basically two types of FFT: Decimation in Time and Decimation in Frequency. We used Decimation in frequency for implementing FFT.
Decimation in frequency:
It is obtained by using a divide and conquer approach. To derive the algorithm, we begin by splitting the DFT formula into two summations, of which one involves the sum over the first N/2 data points and the second involving the sum over last N/2 data points.
So we obtain the following

$$X(k) = \sum_{n=0}^{(N/2)-1} x(n)W_N^{kn} + \sum_{n=N/2}^{N-1} x(n)W_N^{kn}$$

$$= \sum_{n=0}^{(N/2)-1} x(n)W_N^{kn} + W_N^{Nk/2} \sum_{n=0}^{(N/2)-1} x\left(n+\frac{N}{2}\right)W_N^{kn}$$

Substituting $W_N^{kN/2} = (-1)^k$ the above expression can be written as

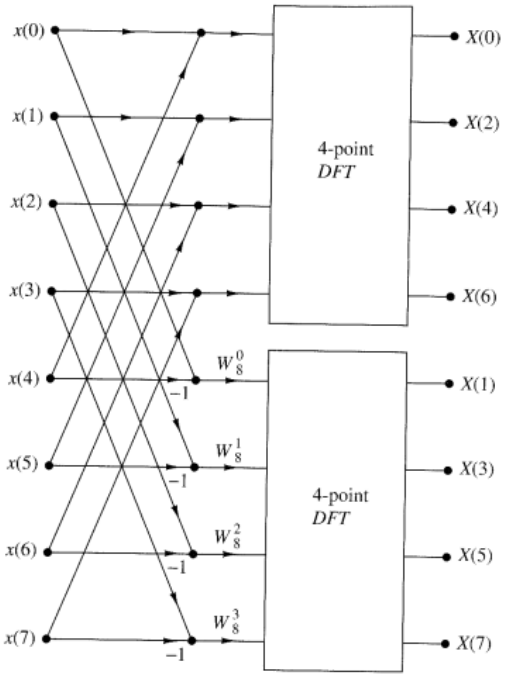
$$X(k) = \sum_{n=0}^{(N/2)-1} \left[x(n) + (-1)^k x\left(n+\frac{N}{2}\right)\right] W_N^{kn}$$

We now split/decimate X(k) into even numbered and odd numbered samples. The radix-2 decimation in frequency algorithm rearranges the DFT into two parts: computation of even-numbered discrete frequency indices and computation of odd-numbered indices and is shown as follows

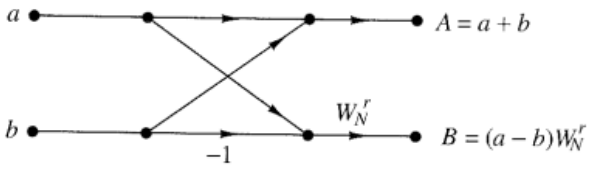
$$
\begin{aligned}
X(2r) &= \sum_{n=0}^{N-1} x(n) W_N^{2rn} \\
&= \sum_{n=0}^{\frac{N}{2}-1} x(n) W_N^{2rn} + \sum_{n=0}^{\frac{N}{2}-1} x\left(n+\frac{N}{2}\right) W_N^{2r\left(n+\frac{N}{2}\right)} \\
&= \sum_{n=0}^{\frac{N}{2}-1} x(n) W_N^{2rn} + \sum_{n=0}^{\frac{N}{2}-1} x\left(n+\frac{N}{2}\right) W_N^{2rn} 1 \\
&= \sum_{n=0}^{\frac{N}{2}-1} \left(x(n) + x\left(n+\frac{N}{2}\right)\right) W_{\frac{N}{2}}^{rn} \\
&= \mathrm{DFT}_{\frac{N}{2}} \left[x(n) + x\left(n+\frac{N}{2}\right)\right]
\end{aligned}
$$

$$
\begin{aligned}
X(2r+1) &= \sum_{n=0}^{N-1} x(n) W_N^{(2r+1)n} \\
&= \sum_{n=0}^{\frac{N}{2}-1} \left(x(n) + W_N^{\frac{N}{2}} x\left(n+\frac{N}{2}\right)\right) W_N^{(2r+1)n} \\
&= \sum_{n=0}^{\frac{N}{2}-1} \left(\left(x(n) - x\left(n+\frac{N}{2}\right)\right) W_N^n\right) W_{\frac{N}{2}}^{rn} \\
&= \mathrm{DFT}_{\frac{N}{2}} \left[\left(x(n) - x\left(n+\frac{N}{2}\right)\right) W_N^n\right]
\end{aligned}
$$

These sequences are computed individually and the subsequent use to compute the N/2 point DFTs are depicted in the following figure.



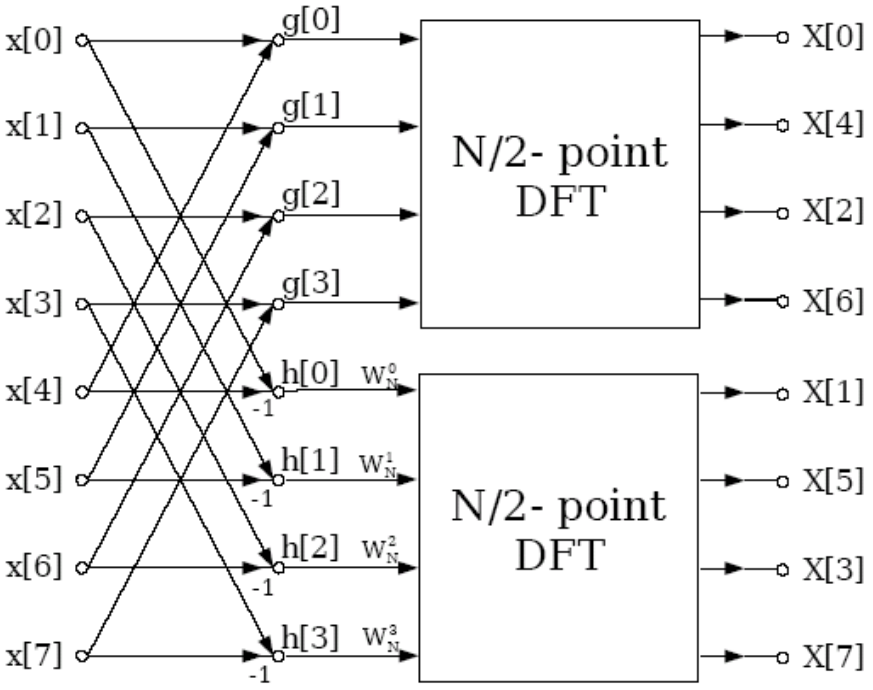
The above figure shows decimation in frequency FFT algorithm for 8-DFT.
This computational procedure can be repeated through decimation of the N/2 point DFTs, X(2k) and X(2K+1). This whole process would require $\log_2 N$ stages of decimation, where each stage involves the N/2butterflies as shown here.



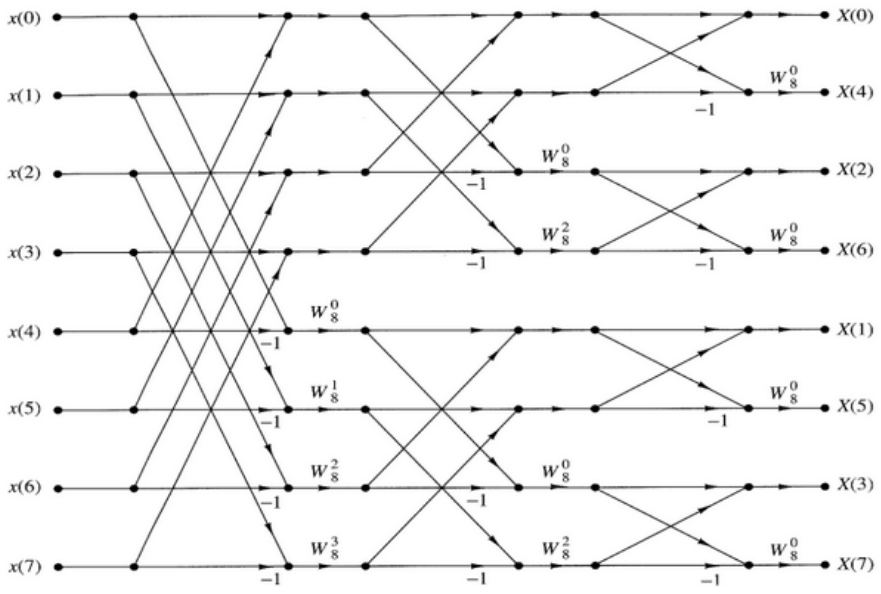
The above figure shows a basic butterfly computation in decimation in frequency FFT algorithm.

The computations of the N-DFT via the decimation in frequency FFT algorithm requires $(N/2)\log_2 N$ complex multiplication and $N\log_2 N$ complex additions.

This is the block diagram we used while coding



An 8-DFT has been illustrated using the decimation in frequency FFT algorithm.



N = 8 point decimation in frequency FFT algorithm

Here we observe that the input data x(n) occurs in natural order, but the output DFT occurs in bit reversed order which can be rearranged later.

Overall FFT algorithm:

The FFT involves separating the N points into smaller groups. We compute the first stage with groups of two coefficients, yielding N/2 blocks, each computing the addition and subtraction of the coefficients scaled by the corresponding twiddle factors (called a “butterfly” for its cross-over appearance).These results are used to compute the next state of N/4 blocks, which will then combine the results of two previous blocks (combining coefficients at this point). This process repeats until we have one main block, with a final computation of all N coefficients.

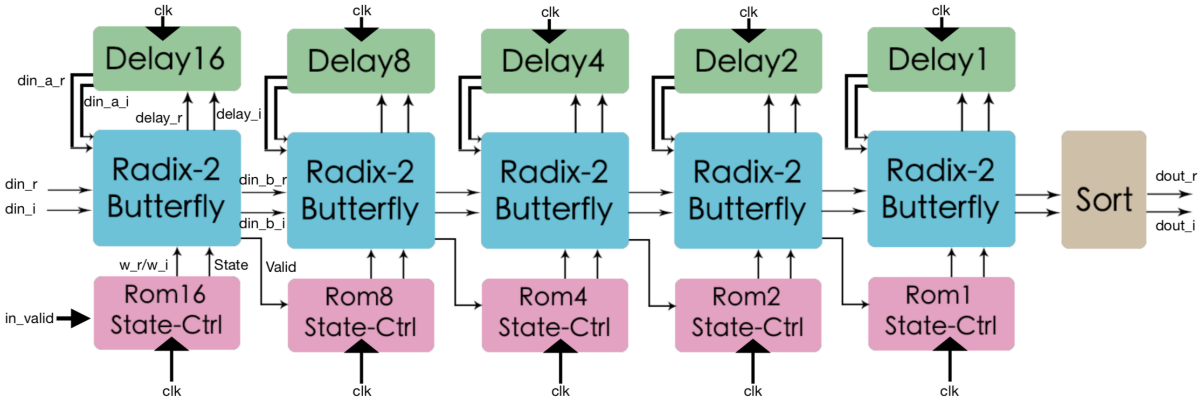
32 - Point FFT

We first made a 32-point pipelined Fast Fourier Transform processor, using single path delay architecture.

We used the following research paper as a reference :
Analysis and implementation of SDF Radix-2 FFT processor using VHDL
[http://www.warse.org/IJATCSE/static/pdf/file/ijatcse144942020.pdf]

Design

Complete Block Diagram



The design is based on the radix2 decimation in frequency algorithm.

Delay Module:

- The delay module is simply a FIFO shift register. It will shift 24 bits every cycle. For delay 16, the shift register size is 16x24 = 384.
- The delay module has 2 functions. First is to load the input while we are in the waiting state and the other is to multiply the twiddle factors stored in ROM with the subtracted input.
- First function : In the waiting state, we cannot do any calculation. For instance, we have to wait for x[16] in the first state to do x[0]+x[16], so x[0]~x[15] will be in the waiting state.
- Second function: we multiply the delay signal, which is the signal we output to the delay module in the first half state, with our twiddle factor.

Radix-2 butterfly:

While entering the first radix-2 butterfly, din is extended to 24bits to match the twiddle factor. The radix-2 butterfly is the core processor of this circuit. It contains three states :

1. Waiting: In the waiting state, we cannot do any calculation. For instance, we have to wait for x[16] in the first state to do x[0]+x[16], so x[0]~x[15] will be in the waiting state.
2. First Half: In the first half, the output will be the summation of two indexes, e.g. x[0]+x[16]. We will output x[0]-x[16] to delay module simultaneously.
3. Second Half: In the last state, we multiply the delay signal, which is the signal we output to the delay module in the first half state, with our twiddle factor. Same as above, the input (din_a_r and din_b_r) will be output to the delay module. The complex number multiplication is transformed from 4 multiplication and 2 summation to 3 multiplication 5 summation.

Rom and State Control:

- The ROM is where the twiddle factors are stored. When it receives the valid signal from the front stage, it sets a counter.
- Based on the counter, it will output a state control output signal to the radix2 butterfly module.
- For the second half state where multiplication takes place, it will provide the needed twiddle factor.

Sort Module

- Since this is decimation by frequency the output will come in scrambled order, we have to correct it.
- Since we know the output signal order, we can simply control the order by directing the signal to a 2D array and place the value in the right place.
- For instance, the 2nd output is X[15], we can store it into result[15].
- It will take 32 cycles to sort. To add on, the input of the sort module is the most significant 16 bits of the output of the last radix-2 butterfly.

Algorithm that we followed

We give the input serially at negative clock edges. The delay block now loads all the 128 inputs. Then it changes state and calculates the sum and difference of the in[128],in[129]... and in[0],in[1]...

It sets the input as valid for the next block and sends the sum and sends the difference to the delay to be multiplied with the twiddle factors.

This delayed difference comes after 128 clock cycles and then is multiplied with the twiddle factor that is stored in the ROM.

This process is carried out by each step of the algorithm where each step calculated the corresponding factor of 2

The first Radix-2 module runs only once as in[0] is added and subtracted with in[129], in[1] with in[130].. and so forth.

But the next Radix-2 module runs twice (And 2X as we go further every stage) as we need to compute FFT of two sets of 128 inputs from the previous Radix module and we again send forth the sums and differences computed in the current Radix module.

The computations in this module starts only in the 128 + 65 = 193rd cycle, as the delay block of the second radix module waits to receive 64 inputs, and starts computing addition and subtractions only once the 65th input is received.

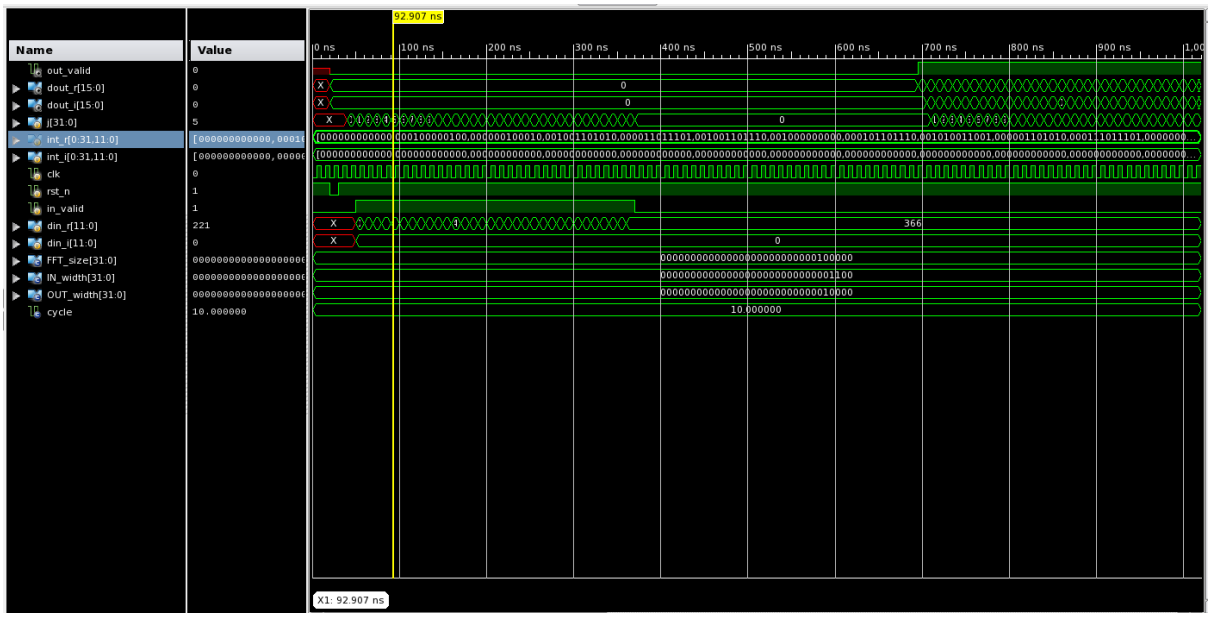
Similar to the previous block, here too we send the sum; and the difference to be multiplied by appropriate twiddle factors to the delay of the next block.

For 32 point FFT, we need five stages and for 256 point FFT we need 8 stages. Each stage consists of a Radix-2 module. Only difference comes in the delay and ram blocks in each stage. For a **simple pipeline FFT processor,** in each stage there are two delay modules and one multiplication with twiddle value.

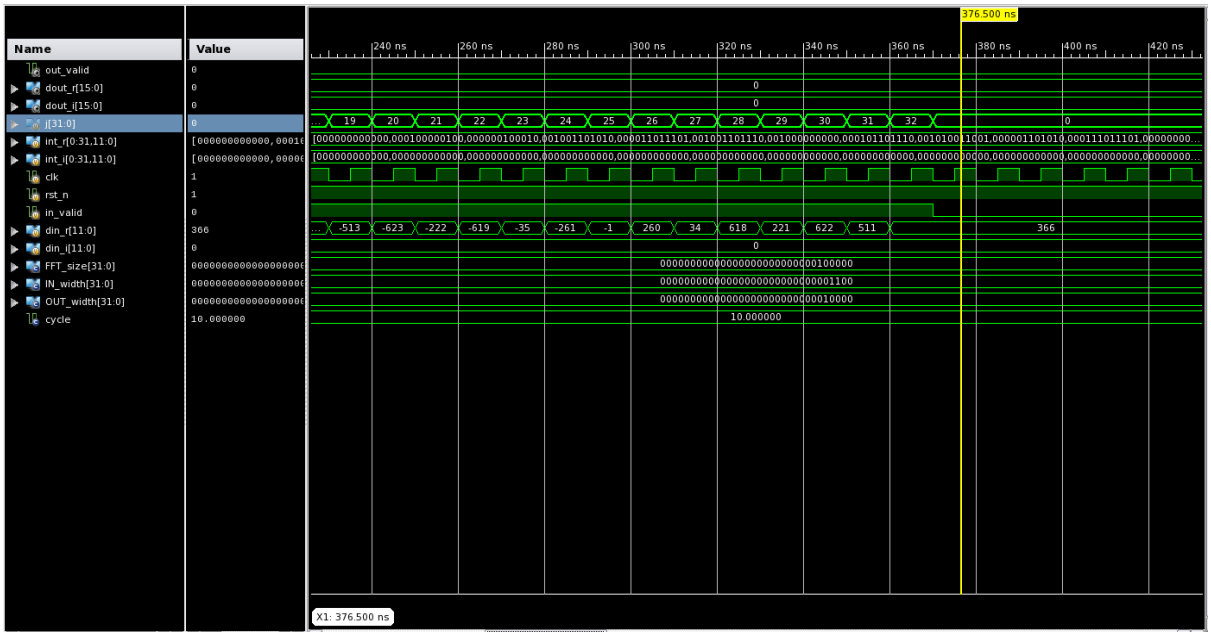
Taking example of the 32 point FFT, a delay-16 module is used to set 16 signals that go to the upper input of Radix-2 module and remaining 16 inputs in lower input of Radix-2. The upper output of Radix-2 will directly go to the next stage. In contrast the lower outputs will come to another delay-4 module and take the multiplication with corresponding twiddle values which was stored in ROM.

RESULTS:

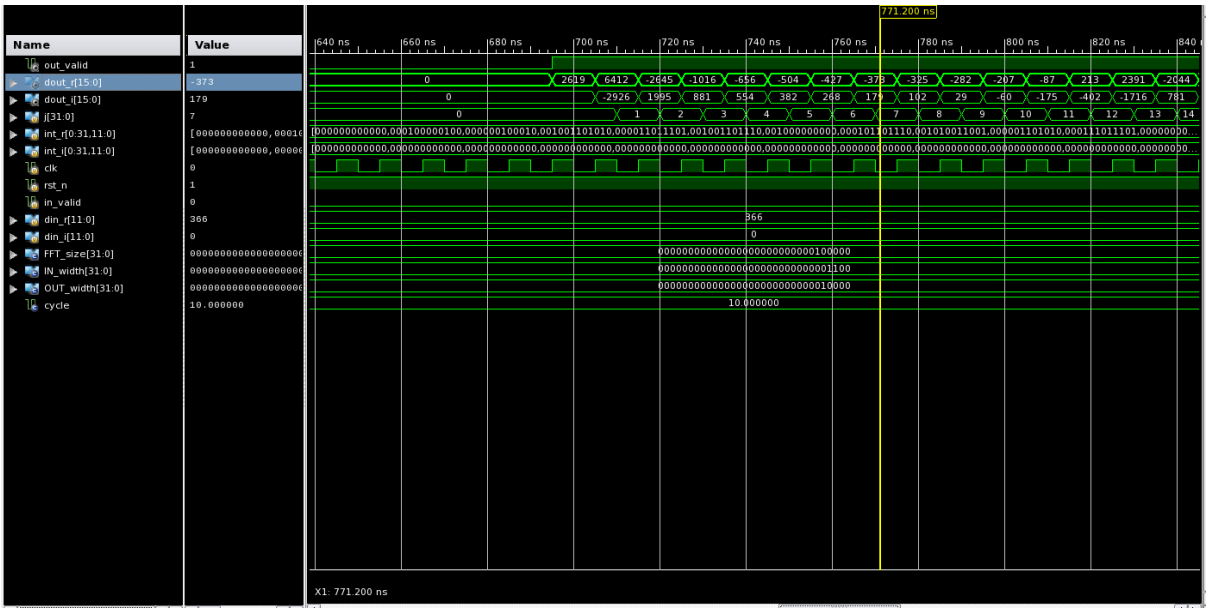
Output of the complete simulation for 32 FFT



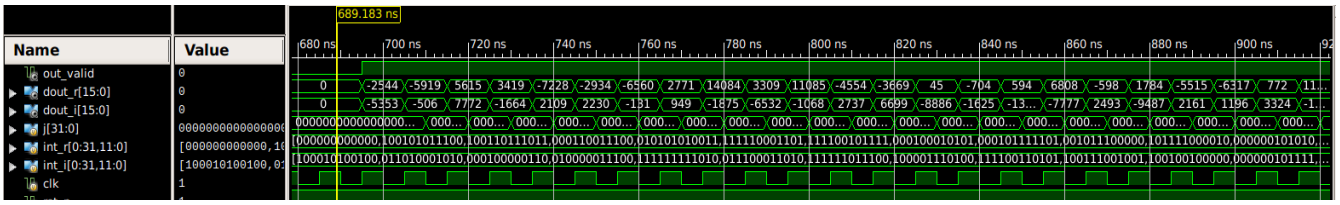
Loading Phase for 32 FFT



Output Phase for 32 FFT



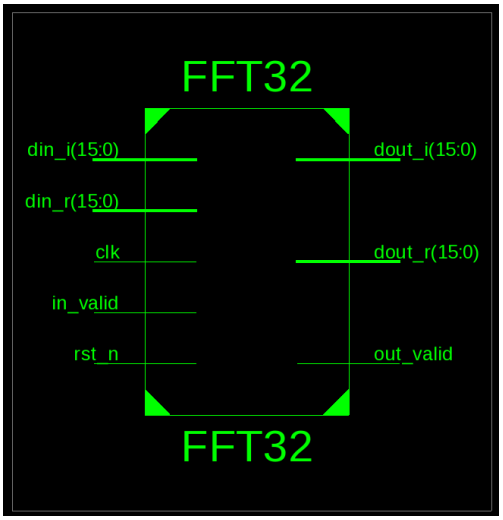
Comparing the results obtained from our code with that from an [online tool to compute FFT.]



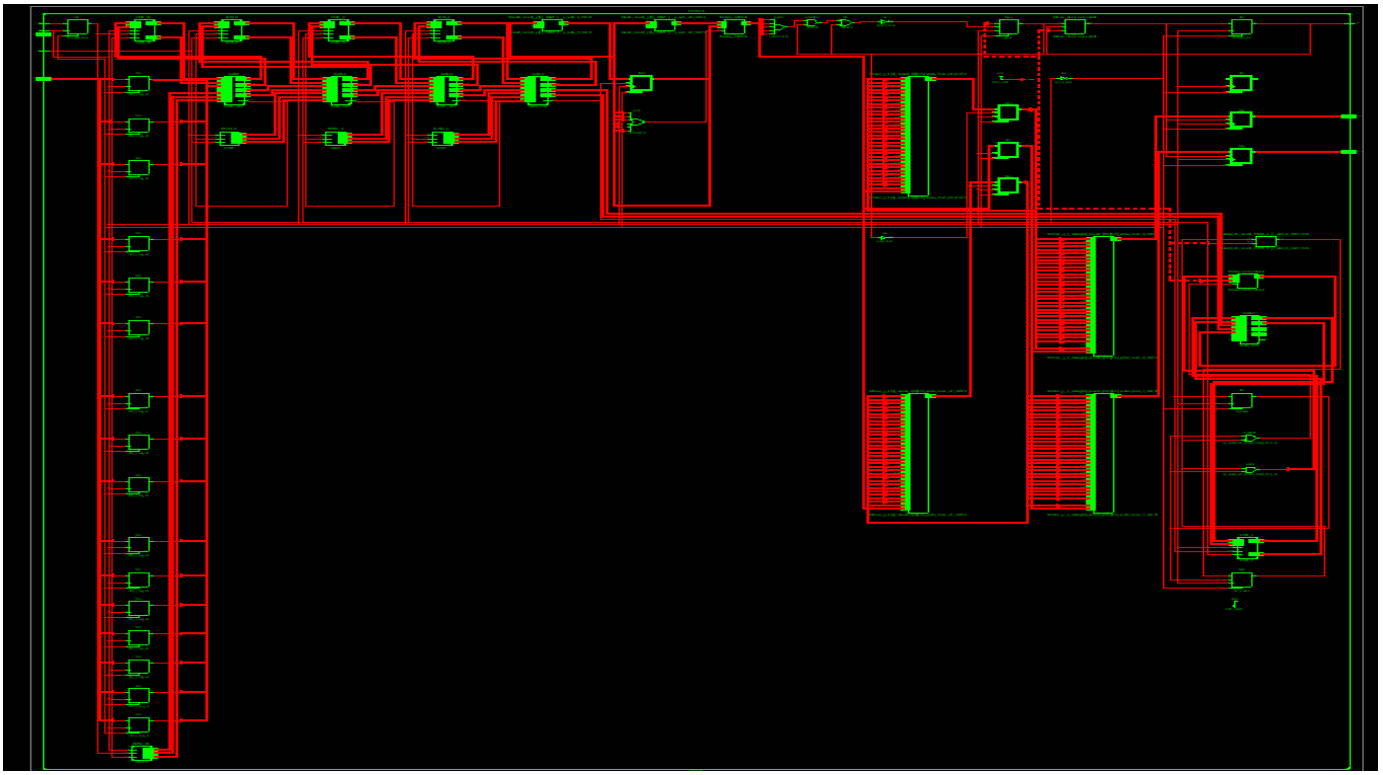
-2544.000000,-5353.000000
-5910.183418,-516.529404
5614.581902,7755.270135
3413.802375,-1661.005573
-7227.376767,2109.790981
-2932.550750,2218.902292
-6550.902187,-150.209627
2767.691386,948.416873
14084.000000,-1875.000000
3301.062390,-6524.053992
11075.139564,-1068.576929

-4548.429407,2729.322718
3668.096829,6700.639313
43.715827,-8872.559691
-706.020618,-1622.089366
596.212179,-13330.544300
6808.000000,-7777.000000
-590.529453,2489.560824
1785.171781,-9469.321359
-5504.882075,2161.927227
-6316.623233,1196.209019
775.997601,3308.722230
1102.813008,-18956.820949

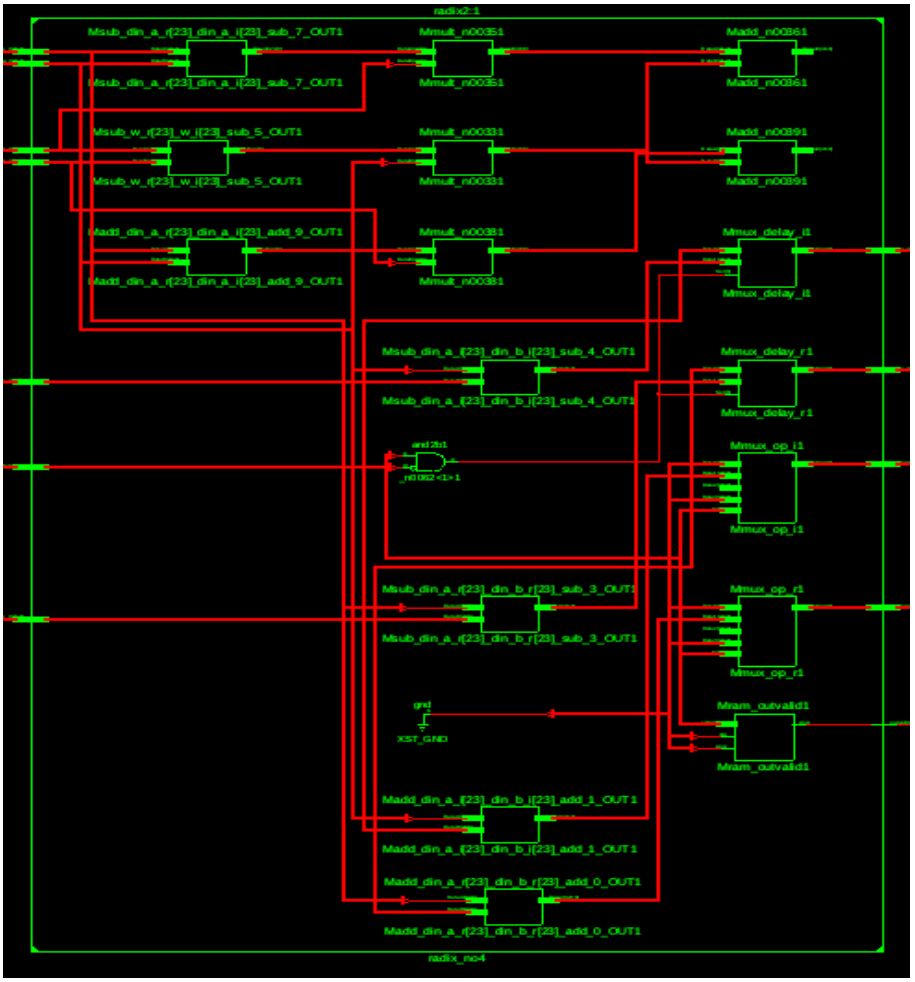
RTL Schematic of 32 FFT



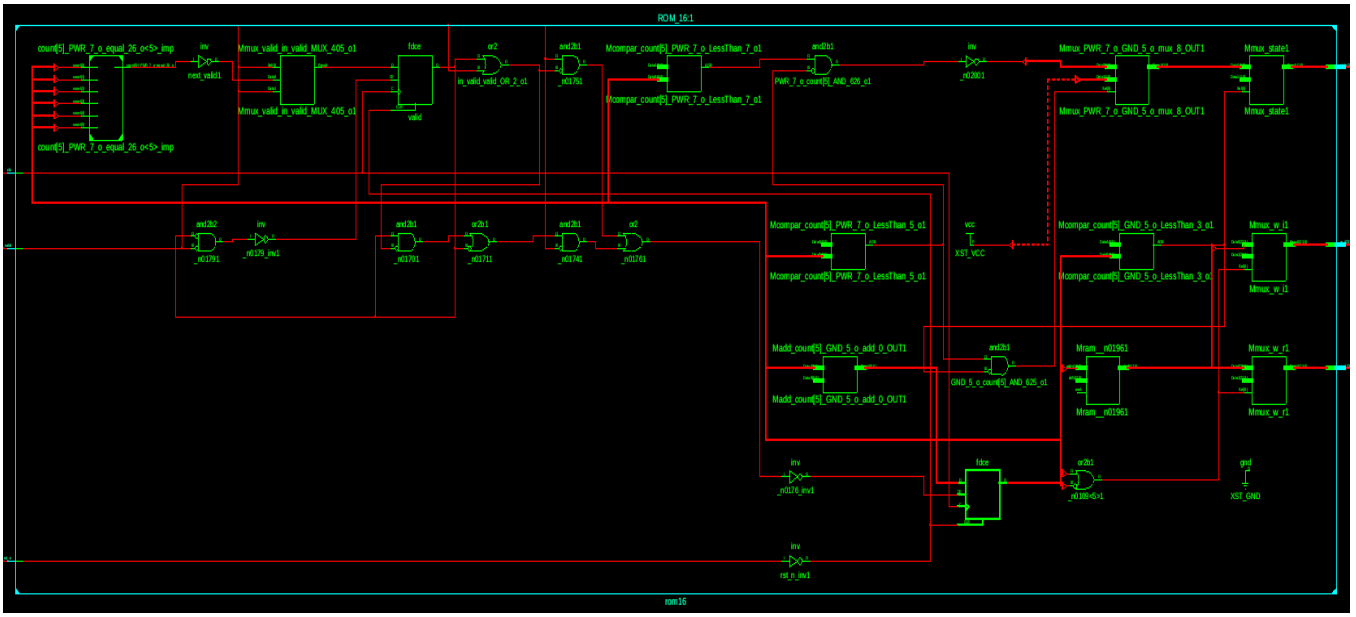
Complete RTL Schematic of 32 FFT



Schematic of the Radix-2 butterfly module



Schematic on the ROM-16 module used in 32-FFT



Timing Details of 32-FFT

```
==============================================================================
Timing constraint: Default period analysis for Clock 'clk'
  Clock period: 41.856ns (frequency: 23.891MHz)
  Total number of paths / destination ports: 25077439511543 / 3676
------------------------------------------------------------------------------
Delay:               41.856ns (Levels of Logic = 48)
  Source:            rom16/count_0 (FF)
  Destination:       result_r_31_15 (FF)
  Source Clock:      clk rising
  Destination Clock: clk rising
```

Device Utilization Summary of 32-FFT obtained from Xilinx

Device Utilization Summary				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Total Number Slice Registers	1,727	33,280	5%	
Number used as Flip Flops	1,723			
Number used as Latches	4			
Number of 4 input LUTs	2,384	33,280	7%	
Number of occupied Slices	1,864	16,640	11%	
Number of Slices containing only related logic	1,864	1,864	100%	
Number of Slices containing unrelated logic	0	1,864	0%	
Total Number of 4 input LUTs	2,384	33,280	7%	
Number used as logic	2,288			
Number used as Shift registers	96			
Number of bonded IOBs	60	519	11%	
IOB Flip Flops	58			
Number of BUFGMUXs	1	24	4%	
Number of DSP48As	41	84	48%	
Average Fanout of Non-Clock Nets	2.65			

Analysis of Power Distribution from Xpower Analyser

A	B	C	D	E	F	G	H	I	J	K	L	M	N
Device			On-Chip	Power (W)	Used	Available	Utilization (%)		Supply	Summary	Total	Dynamic	Quiescent
Family	Spartan3adsp		Clocks	0.001	1	---	---		Source	Voltage	Current (A)	Current (A)	Current (A)
Part	xc3sd1800a		Logic	0.000	2382	33280	7		Vccint	1.200	0.043	0.001	0.042
Package	fg676		Signals	0.000	5785	---	---		Vccaux	2.500	0.025	0.000	0.025
Temp Grade	Commercial		DSPs	0.000	41	84	49		Vcco25	2.500	0.000	0.000	0.000
Process	Typical		IOs	0.000	60	519	12						
Speed Grade	-4		Leakage	0.114							Total	Dynamic	Quiescent
			Total	0.115					Supply	Power (W)	0.115	0.001	0.114
Environment													
Ambient Temp (C)	25.0		Thermal	Properties	Effective TJA (C/W)	Max Ambient (C)	Junction Temp (C)						
Use custom TJA?	No				15.9	83.2	26.8						
Custom TJA (C/W)	NA												
Airflow (LFM)	0												
Characterization													
PRODUCTION	v1.1,06-26-09												

Power Summary from the Synthesis Report

2. Summary

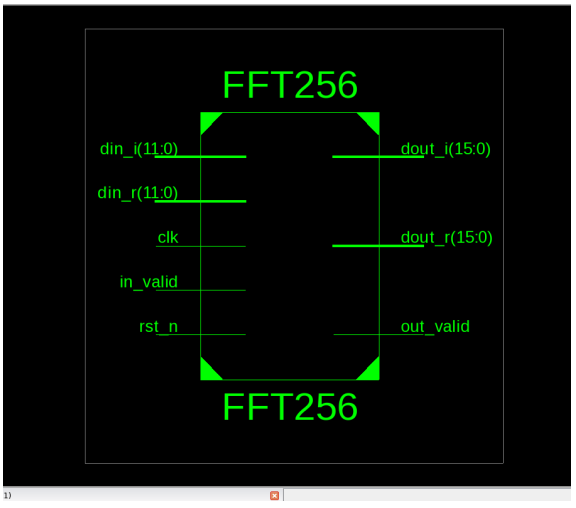
2.1. On-Chip Power Summary

On-Chip Power Summary				
On-Chip	Power (mW)	Used	Available	Utilization (%)
Clocks	0.88	1	---	---
Logic	0.00	2382	33280	7
Signals	0.00	5785	---	---
IOs	0.00	60	519	12
DSPs	0.00	41	84	49
Quiescent	114.09			
Total	114.97			

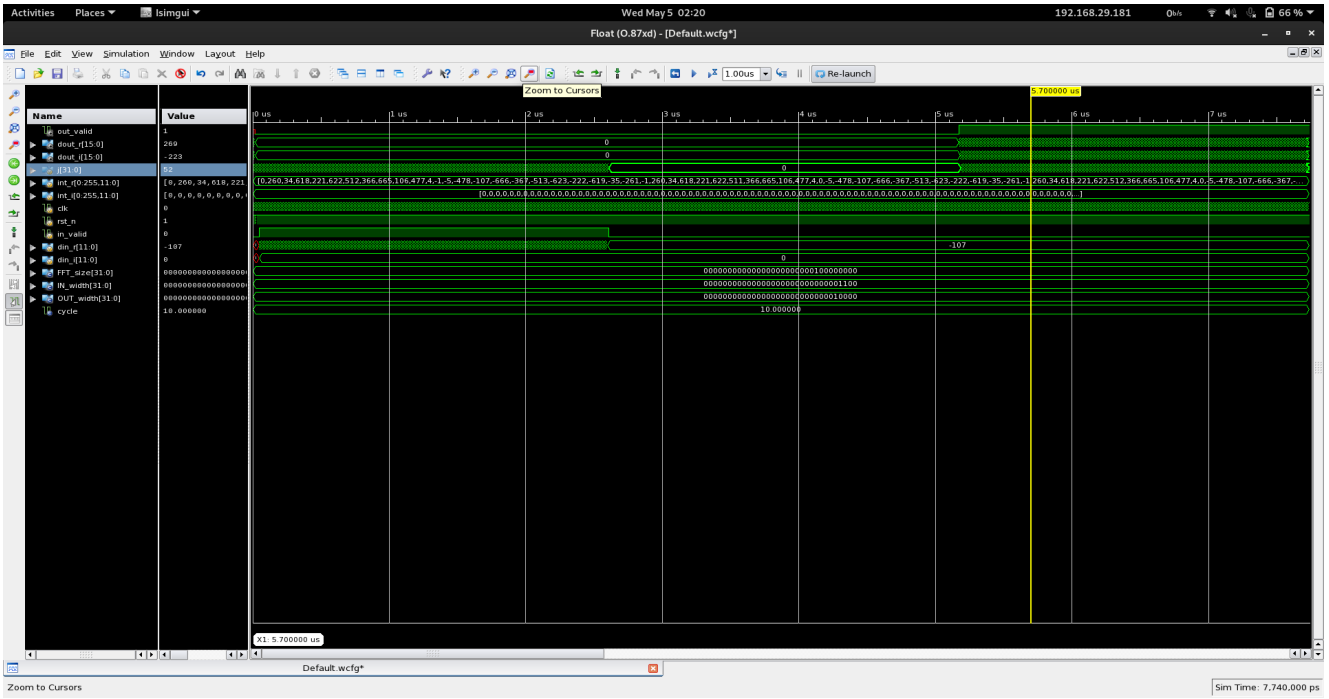
2.2. Thermal Summary

Thermal Summary	
Effective TJA (C/W)	15.9
Max Ambient (C)	83.2
Junction Temp (C)	26.8

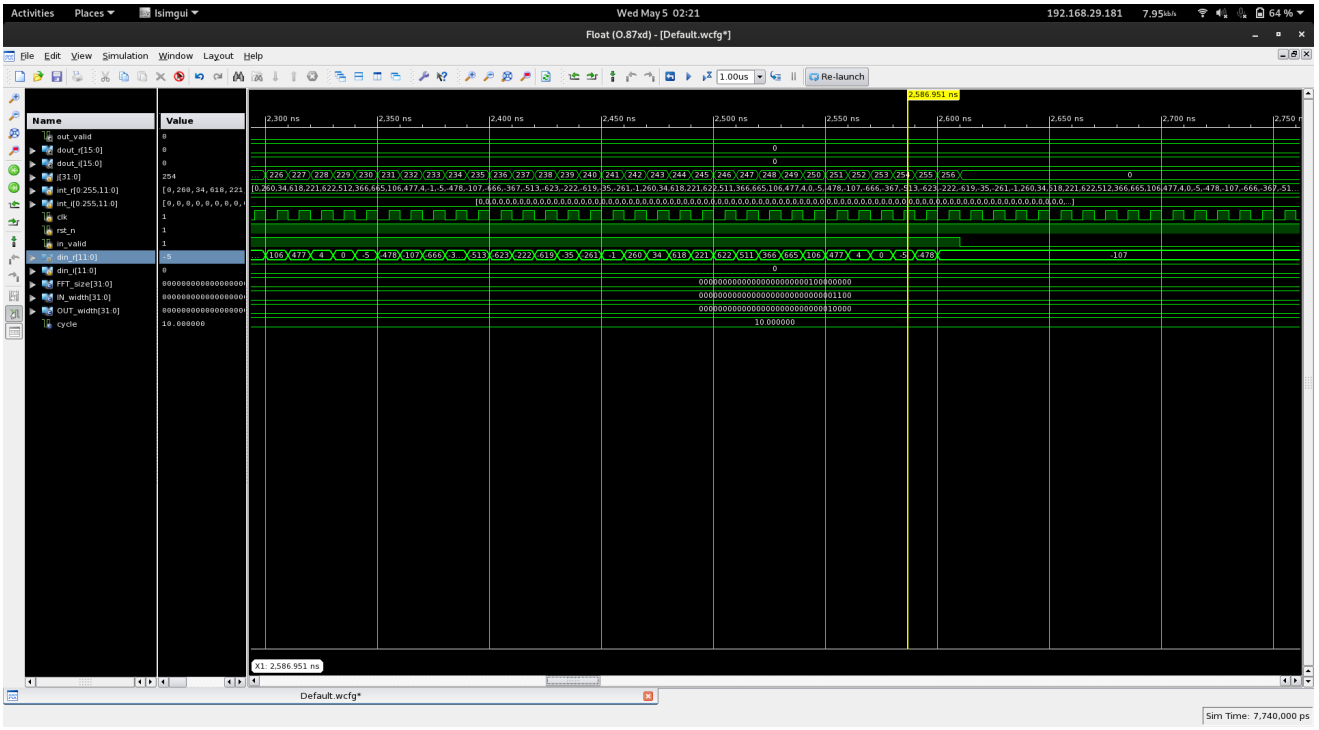
RTL Schematic for 256-FFT



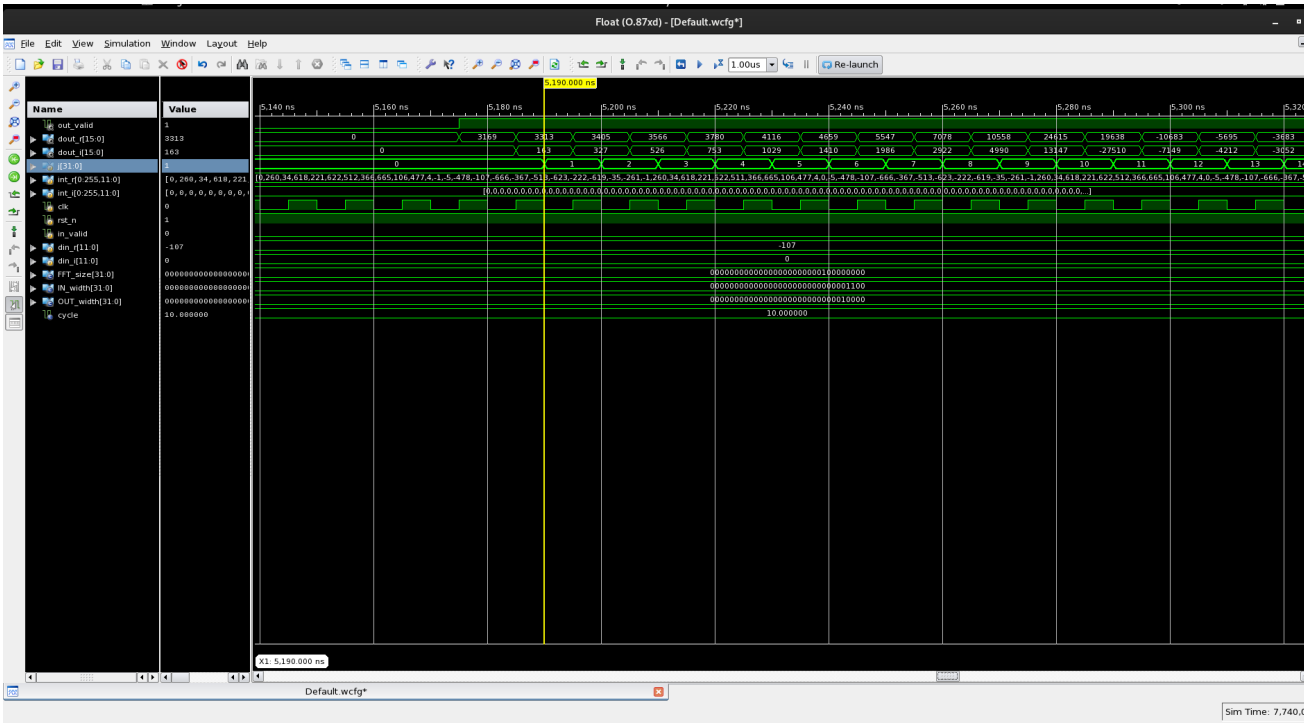
Output of the complete simulation for 256-FFT



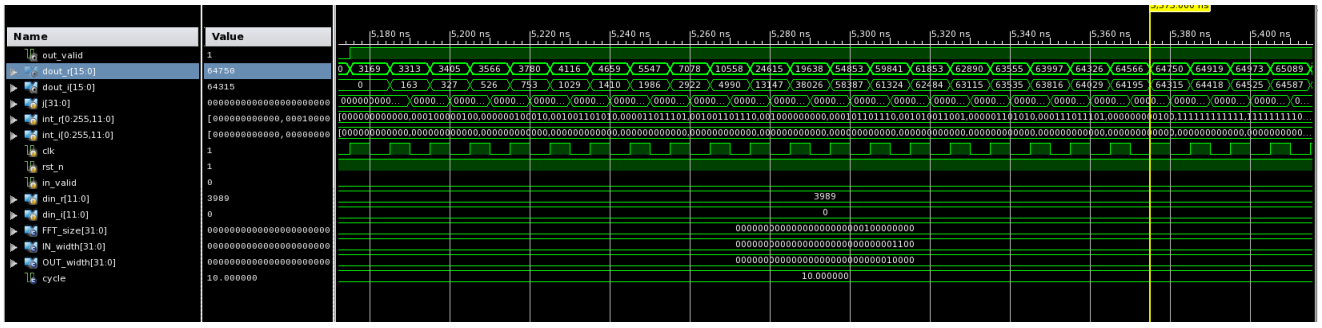
Loading Phase for 256-FFT



Output Phase for 256-FFT



Comparing the results obtained from our code with that from an [online tool to compute FFT.]



3169.000000,0.000000
3325.260415,160.079226
3410.162279,333.460743
3556.147457,526.251365
3782.969592,751.208822
4130.316782,1027.677615
4660.332352,1406.969313
5525.740736,1969.674471

7075.974434,2921.279827
10557.371795,4983.544600
24601.874342,13135.813791
-45858.491164,-27493.551722

Performing 2D FFT on images using the FFT module:

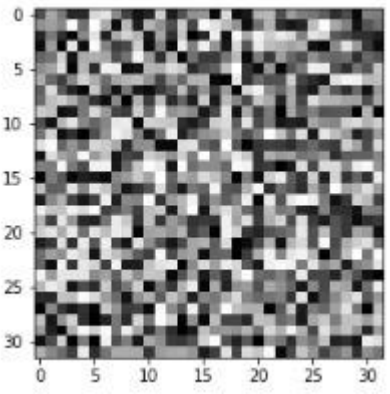
Since images are 2D type objects and here we are calculating only 1D FFT, we need a way to use 1D FFT to perform 2D FFT. The algorithm for the same is as follows:

- First perform 1D FFt in each row and store the output. This converts the real values of the input image into complex values.
- Then perform 1D FFT on each column obtained from the previous step. This is a Complex to complex conversion.

One point to note here is that this cannot be directly applied because verilog cannot load images. So we use python and cv2 (from OpenCV) to generate a random image and save it in .bmp format. Then use a matlab script to convert it into .hex format. We then read the image (32x32) as a (1x1024) flattened array in verilog ad then perform the above algorithm.

This the output obtained in Xilinx when the image shown is given as input

```
ISim>
# run 100.00us
-4024 0
-1235 -71
698 -1286
3868 171
1231 -1283
1242 -1150
-2171 2020
1317 1529
-1737 -3375
-133 1214
2858 -957
-1426 -1018
-1690 1261
-2964 -939
534 -430
-1965 -214
4886 0
-1965 211
537 430
-2970 940
-1690 -1262
-1423 1016
2857 955
-136 -1218
-1737 3375
1314 -1527
-2171 -2020
```

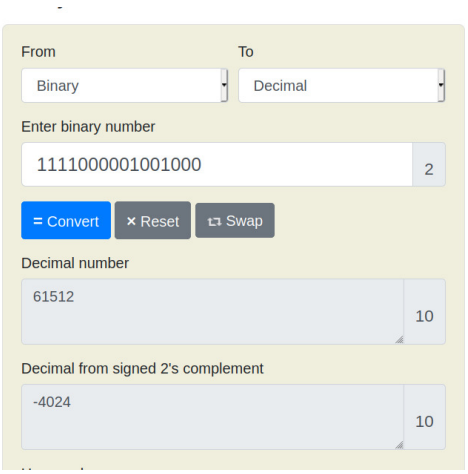


The following shows the FFT output for the same image computed in python.

```
1 from numpy.fft import fft2, ifft2,fft
2 #  fft(tem[0,:])
3 fft2(tem)[:,0]
```

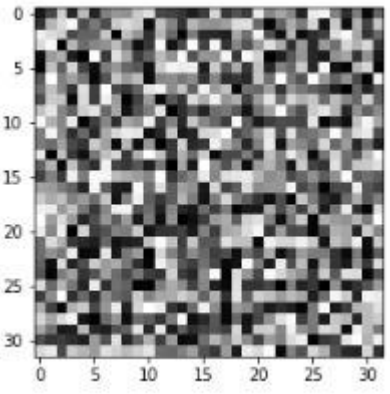
```
array([127048.          +0.j        ,  -1231.88531888  -72.04908869j,
         699.85139534-1283.62876995j,   3863.07771124 +172.53904917j,
        1231.17550315-1282.60660172j,   1246.10762721-1149.40460618j,
       -2165.97045459+2019.30978146j,   1313.67256913+1526.70728926j,
       -1737.        -3375.j        ,   -135.73476256+1215.19882978j,
        2853.61302349 -957.35039032j,  -1423.89581308-1015.33218231j,
       -1689.17550315+1261.39339828j,  -2964.33079699 -938.78968066j,
         536.50603576 -428.28894173j,  -1963.01121606 -212.95870187j,
        4886.          +0.j        ,  -1963.01121606 +212.95870187j,
         536.50603576 +428.28894173j,  -2964.33079699 +938.78968066j,
       -1689.17550315-1261.39339828j,  -1423.89581308+1015.33218231j,
        2853.61302349 +957.35039032j,   -135.73476256-1215.19882978j,
       -1737.        +3375.j        ,   1313.67256913-1526.70728926j,
       -2165.97045459-2019.30978146j,   1246.10762721+1149.40460618j,
        1231.17550315+1282.60660172j,   3863.07771124 -172.53904917j,
         699.85139534+1283.62876995j,  -1231.88531888  +72.04908869j])
```

It is observed that the values obtained are similar except the first bit. This is because the first bit 127048 is a 17 bit number and so there was an overflow of bits resulting in the output to go to -4096 which is shown below.



The following shows output of 2D FFT performed on a different image (but with a smaller pixel range) that was also generated randomly. This also is obtained in Xilinx.

```
Console
Finished circuit initialization process.
ISim>
# run 100.00us
25232 0
342 -57
-80 -260
-89 -9
274 78
-18 92
171 138
516 -177
312 56
63 196
383 -80
-208 -728
243 312
361 -285
-155 62
-339 -686
-636 0
-338 684
-155 -63
361 285
243 -313
-209 727
383 78
63 -197
312 -56
516 176
171 -139
-18 -92
274 -79
-88 8
-80 259
343 57
292 179
```



These are the values obtained when FFT was computed on the same image using python. We can see that the values obtained are very similar to those in

```
[25232.          +0.j           342.94216322 -56.96727809j
 -79.96552472-259.22549754j   -88.15485208  -8.84980339j
 274.55634919 +78.16147161j   -17.84631983 +91.79175446j
 171.51218372+138.63196387j   516.11510854-176.82911033j
 312.          +56.j           63.40706742+196.71248368j
 383.05340019 -79.04213432j  -207.56682615-726.69308378j
 243.44365081+312.16147161j   360.86742121-284.43723873j
-154.60005918 +63.10040426j  -337.76376232-684.52828118j
-636.          +0.j          -337.76376232+684.52828118j
-154.60005918 -63.10040426j   360.86742121+284.43723873j
 243.44365081-312.16147161j  -207.56682615+726.69308378j
 383.05340019 +79.04213432j    63.40706742-196.71248368j
 312.          -56.j          516.11510854+176.82911033j
 171.51218372-138.63196387j   -17.84631983 -91.79175446j
 274.55634919 -78.16147161j   -88.15485208  +8.84980339j
 -79.96552472+259.22549754j   342.94216322 +56.96727809j]
```