1D FFT using a DSP48 IP on FPGA

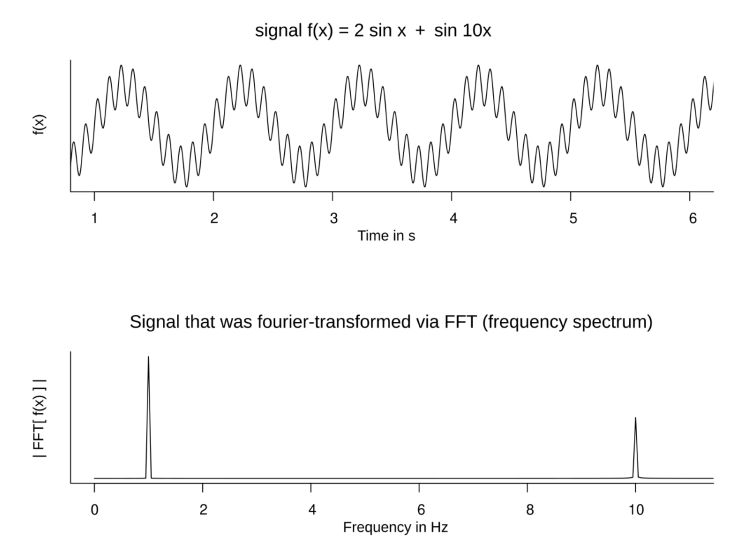
Team Members:

1. A. Nishith – IMT2022556
2. K. S. V. Rohit – IMT2022576
3. A. V. N. Lokesh – IMT2022577

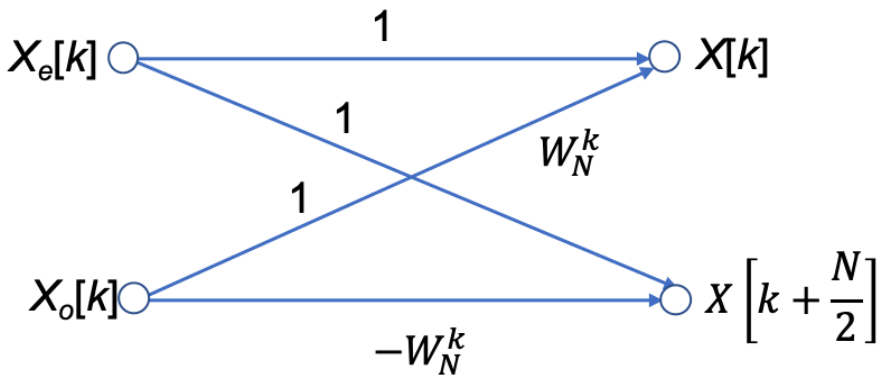
Introduction:

We have implemented a 128-point Fast Fourier Transform (FFT) using the Radix-2 Decimation in Time (DIT) algorithm (Cooley-Tukey algorithm) with DSP48 IP blocks on FPGA.

A Fast Fourier transform (FFT) is an algorithm that computes the Discrete Fourier Transform (DFT) of a sequence. The ordinary DFT algorithm is obtained by decomposing a sequence of values into components of different frequencies and has a time complexity of $O(n^2)$. An FFT rapidly computes such transformations by factorizing the DFT matrix into a product of sparse (mostly zero) factors. One such algorithm for FFT is the Cooley-Tukey algorithm, which re-expresses the discrete Fourier transform (DFT) of an arbitrary composite size $N=N_1N_2$ in terms of N_1 smaller DFTs of sizes N_2, recursively, to reduce the computation time to O (N log N).



A radix-2 decimation-in-time (DIT) FFT is the simplest and most common form of the Cooley–Tukey algorithm. Radix-2 DIT divides a DFT of size N into two interleaved DFTs of size N/2 with each recursive stage as shown in the image above. The last iteration leads to 2-point FFTs (butterfly stage) that can be easily calculated by solving 4 multiplications and 2 additions. We used the DSP48 IP for multiplications.



Butterfly Diagram

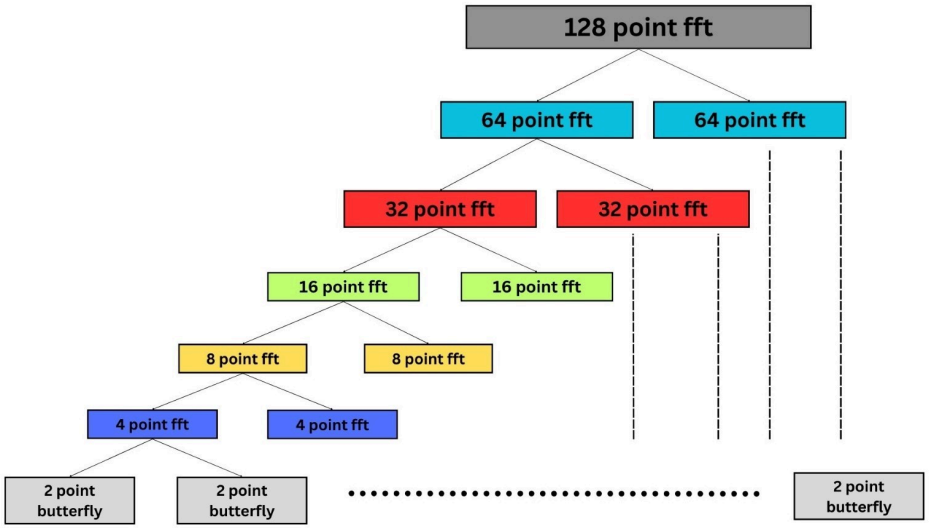
As you can see from the above diagram, in a single 2-point FFT stage (butterfly stage), there are 2 twiddle factors that are multiplied with the previous stage inputs and the respective sums are the outputs.

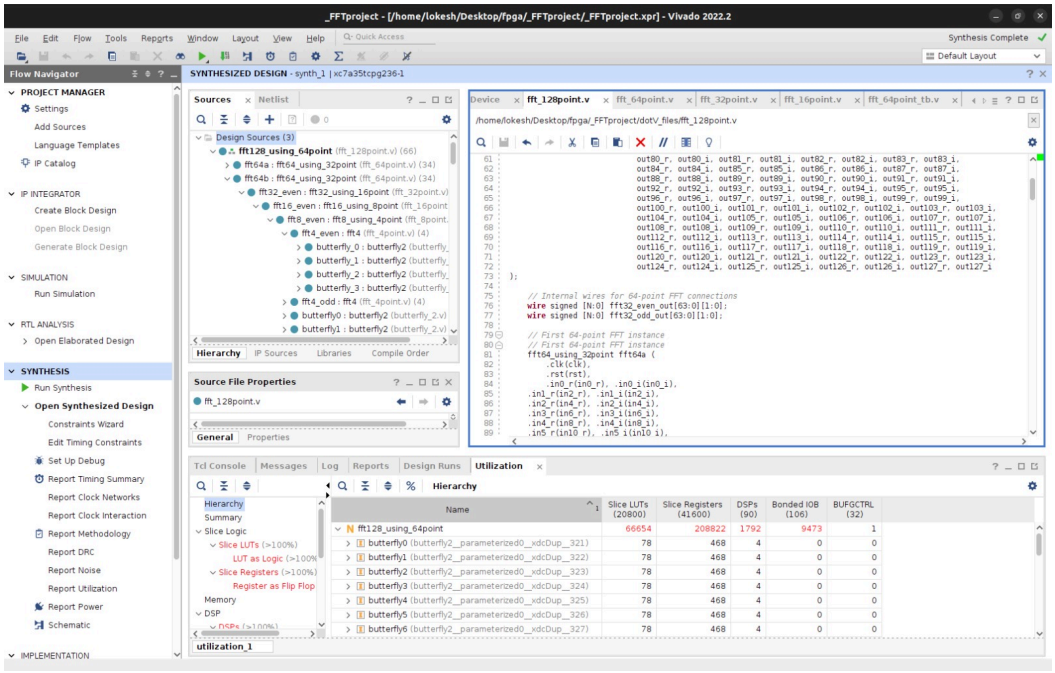
Methodology:

We are using a Q12.6 representation as the DSP48 IP inputs are to be of 18 bits. Since we want to have precision, we use the last 6 bits for fractional part representation and the first 12 bits for whole number part. As a result, all the inputs are multiplied by 64 when being given and the outputs are also received 64 times th actual output.

Approach 1

First, we tried to implement the 128-point FFT by factorizing it into two 64-point FFTs of even index and odd index elements and then performing butterfly stage on the i^{th} output of the even FFT and the i^{th} output of the odd FFT, with the twiddle factor being the multiplier, the even FFT output being the multiplicand and the odd FFT output being the addend. Similarly, the 64-point was factorized into two 32-point FFTs and so on until the 2-point FFT stage.

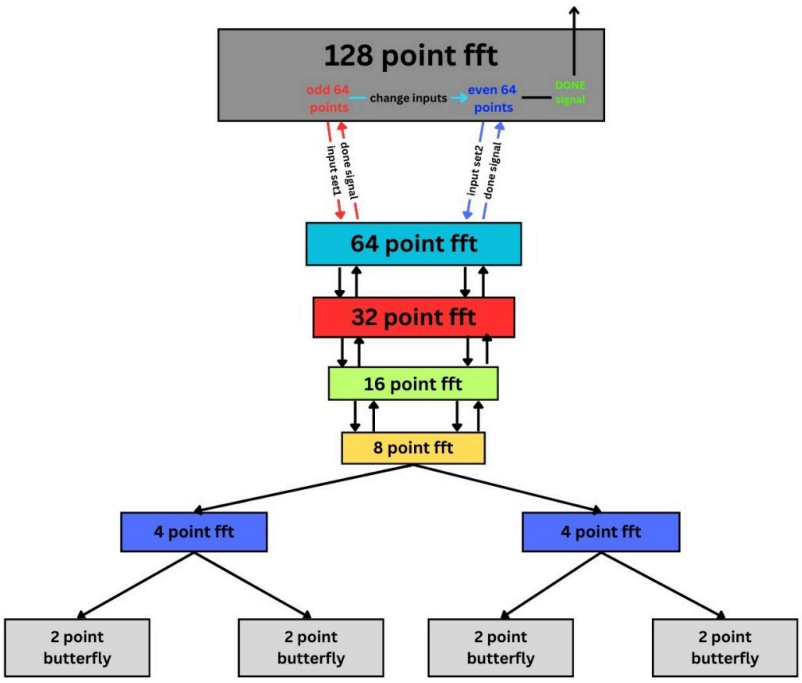




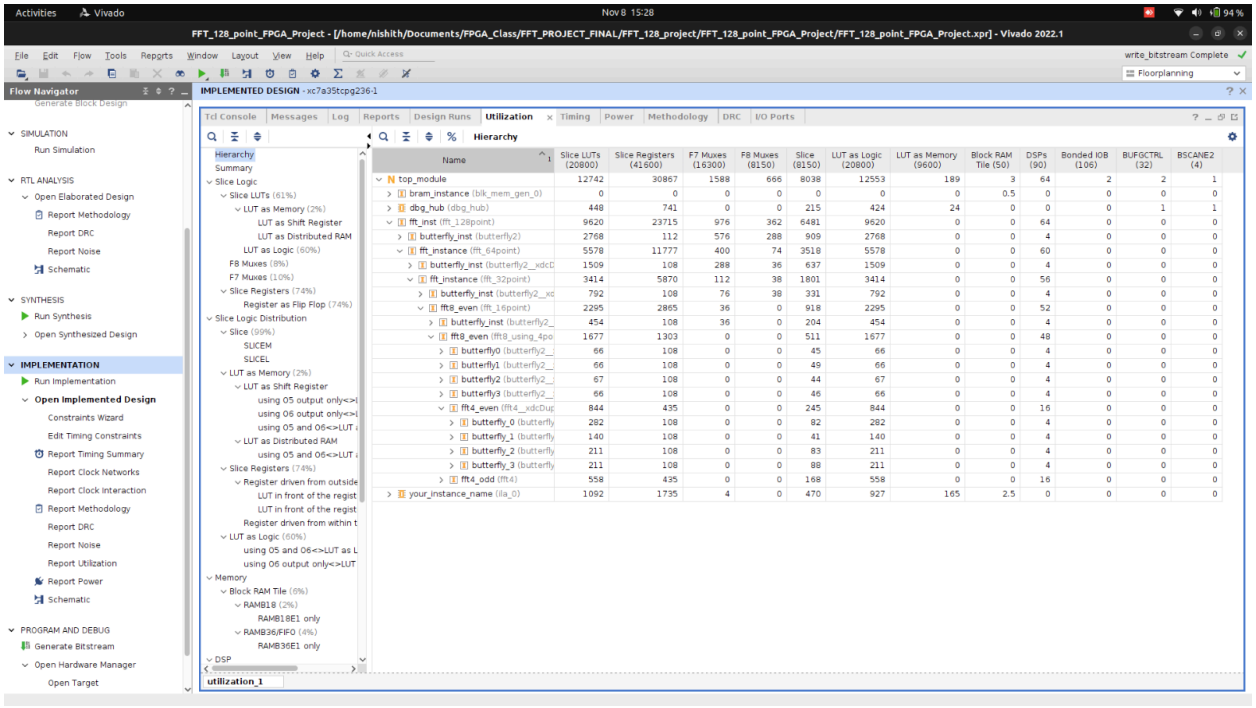
This approach gave us the results we needed but the resource utilization after synthesis step was exceeding 100% on the Basys3 FPGA board as shown in the above utilization report. Since more than 90 DSPs were being used, it would have led us to not being able to implement on the Basys3 board. We identified the reason for this being the DSPs instantiated parallelly causing more DSP resources to be used.

Approach 2 – Multiplexing of the DSPs (Resource Sharing)

The solution we came up with for the previous issue was to use as many DSPs as possible simultaneously in one stage and then reuse them for the next further stages (multiplexing of DSPs). We identified that an 8-point FFT utilized 48 DSPs. So, the largest possible stage that can be implemented at once is the 8-point FFT. So, 16-point FFT can be implemented as 2 8-point FFT stages with their outputs being passed through butterfly operations, giving us the final outputs. Similarly, 32-point, 64-point and 128-point can be implemented as shown in the image below.



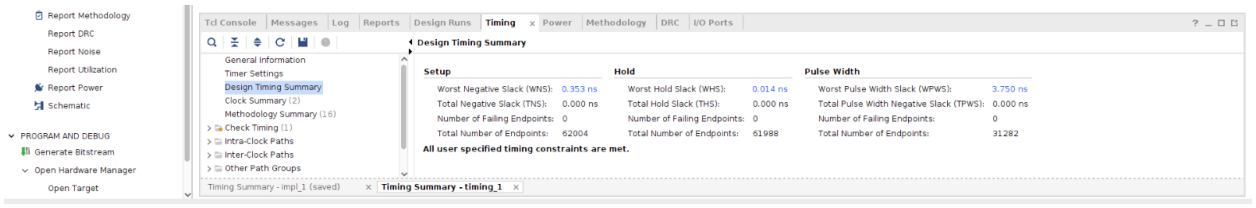
For 16-point, we divide it into 2 stages of 8-point FFTs and start the first FFT. Once the first FFT is completed, we use a done signal that starts the second FFT. Once the second FFT sends the done signal, we perform butterfly operations on their corresponding outputs and achieve the final outputs. Similarly, the 32-point, 64-point and 128-point can be split into 2 stages each and butterfly operations on their outputs give us the correct outputs.



As we double the size of the FFT from 8 to 16, we need only one extra butterfly operation, adding 4 more DSPs and similarly as we double up more to 32, 64 and 128, it adds up 4 every time we double up. So, this gives a utilization of a maximum of 64 DSPs (48 + 4*4) as shown in the above image of the Resource Utilization Hierarchy. This is our final approach.

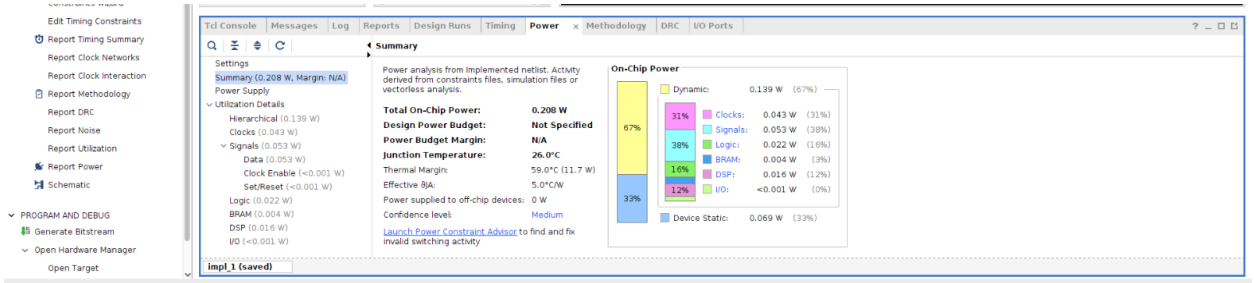
Results:

Time period of 1 clock cycle = 10ns as shown in the above image.

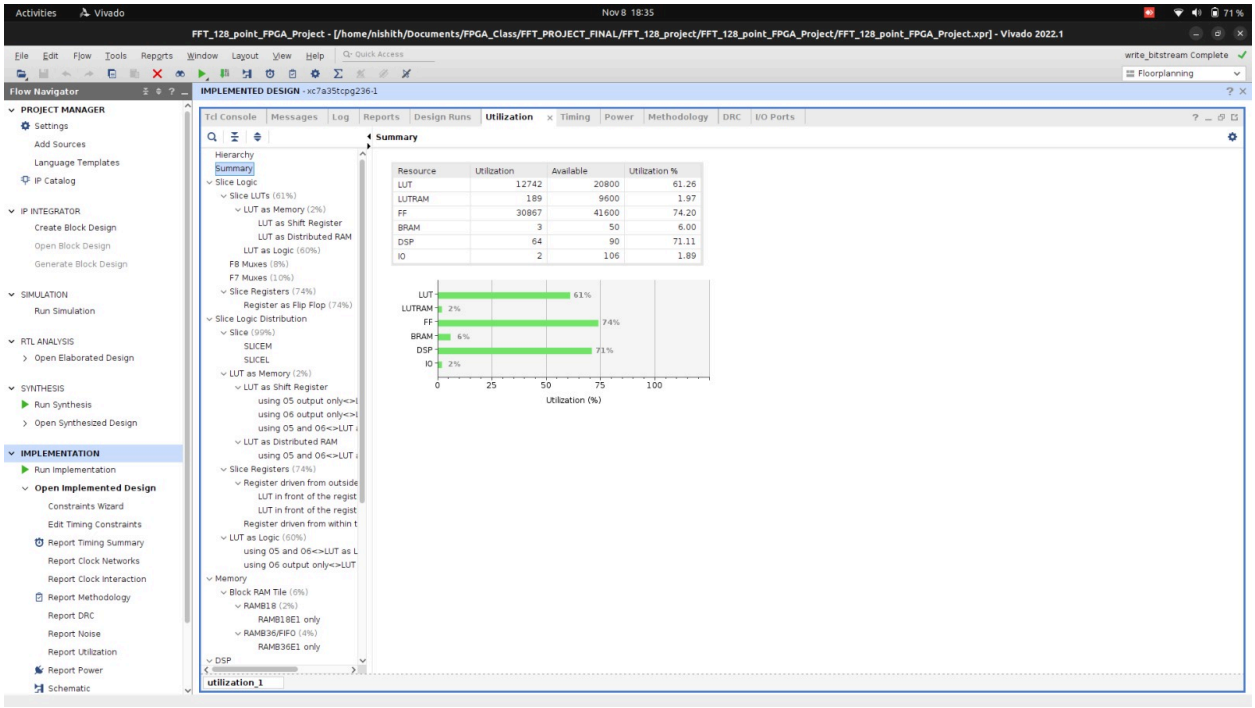


Timing Summary that shows the Worst Negative Slack

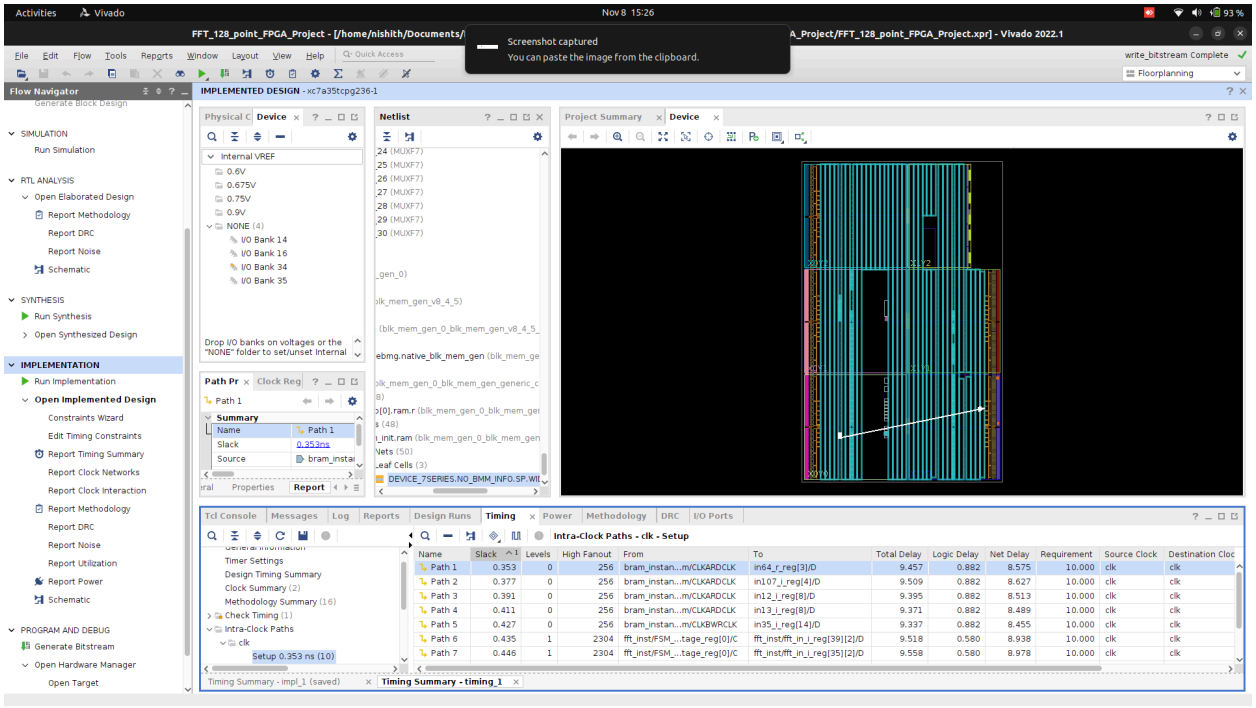
Maximum clock frequency = $(1/(T - WNS))$ = $1/(10ns - 0.353ns)$ = 103.659 MHz



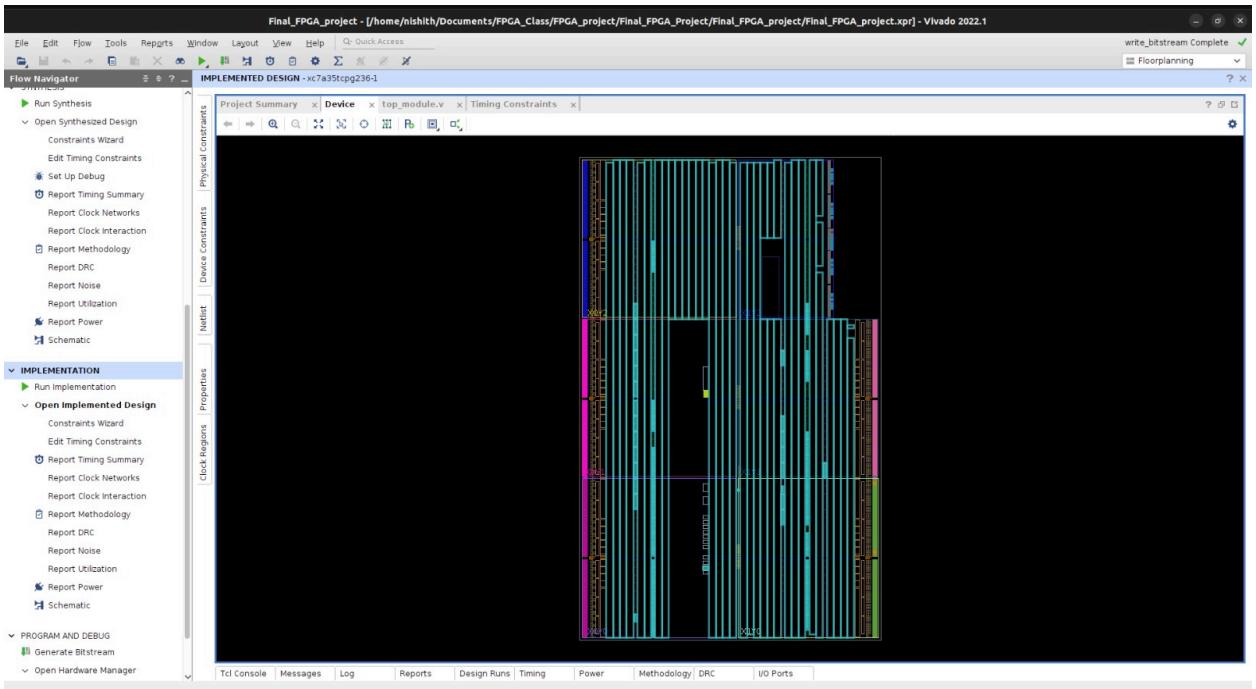
Power Report Summary



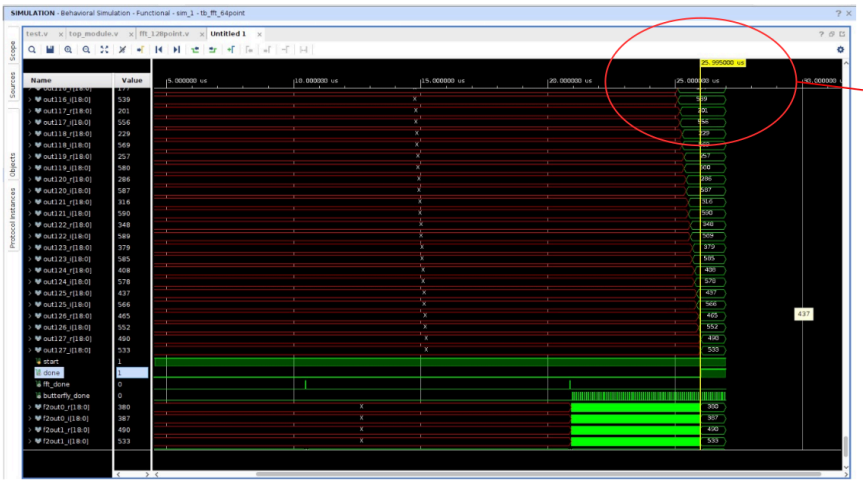
Resource Utilization Summary



The white line denotes the critical path of the circuit

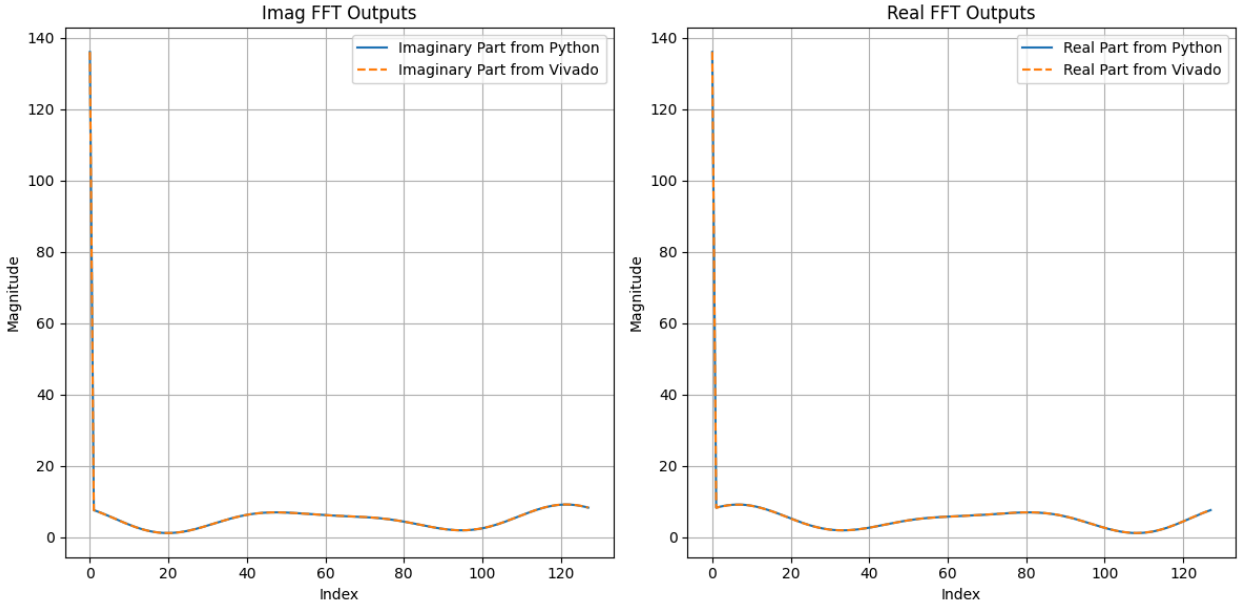


Layout Diagram

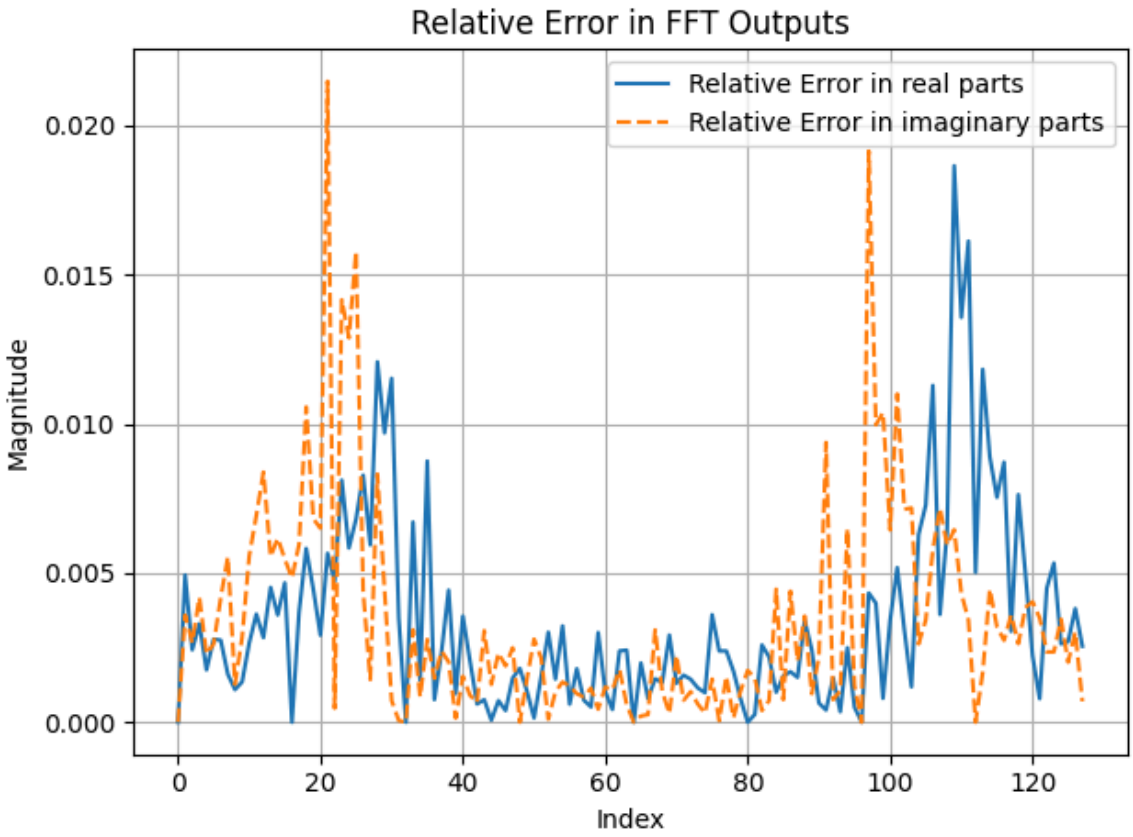


The done signal is high after 25.995us, which is almost 2566 clock cycles (with a 100MHz clock).

The outputs of behavioral simulation of the Verilog code, displaying the outputs in Q12.6 notation, in decimal format. From the image we can infer that the latency of computation is 2566 clock cycles.

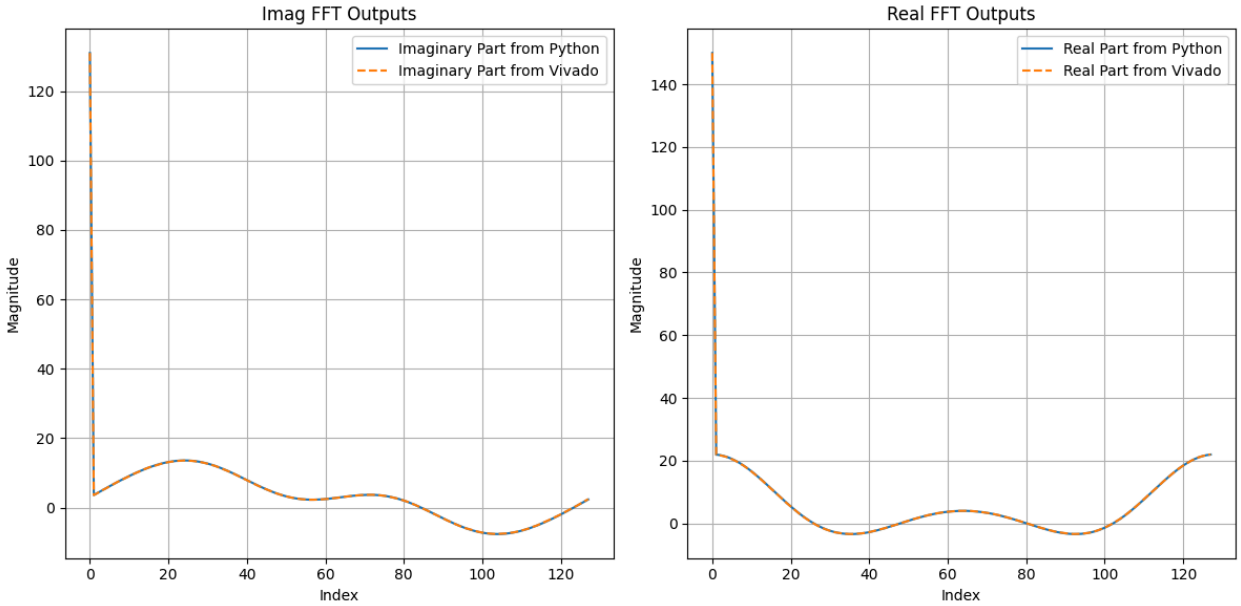


Comparison of the real and imaginary parts of the FFT Outputs from Vivado Behavioral Simulation and Python using the same inputs

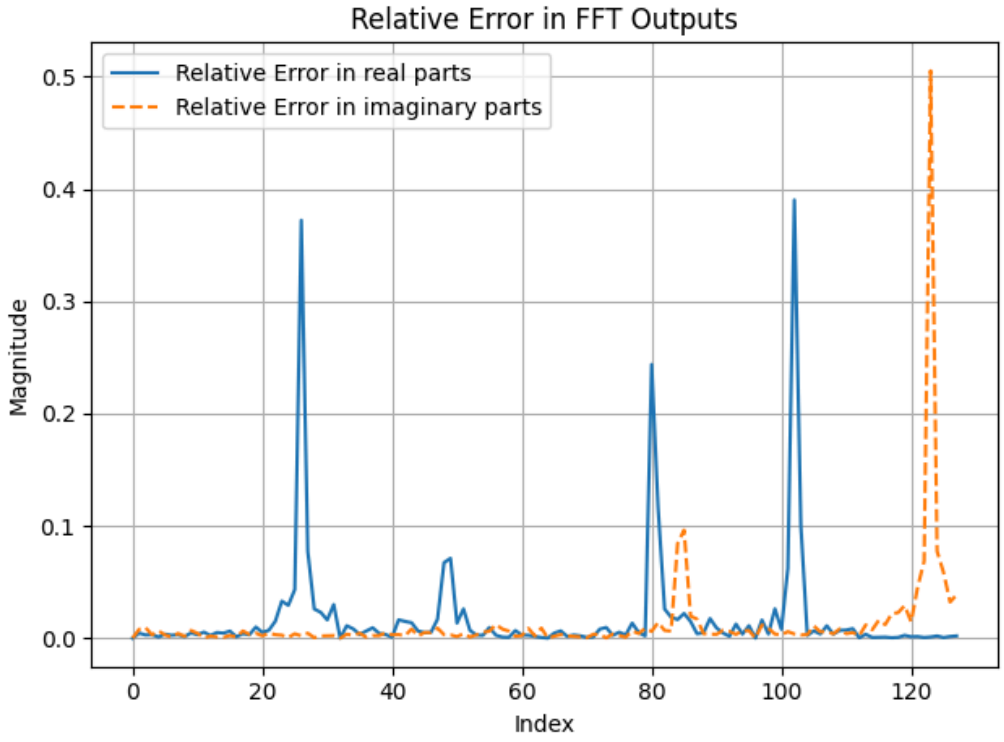


Absolute Relative Error in the outputs of FFT from Vivado Behavioral Simulation compared to Python for the same Inputs

The outputs of the FFT block in Q12.6 representation in decimal format, displayed in the ILA.



Comparison of the real and imaginary parts of the FFT Outputs from
From FPGA using ILA and Python using the same inputs



Absolute Relative Error in the outputs of FFT from Vivado Behavioral Simulation
compared to Python for the same Inputs