Faculty of Engineering
Cairo University

Submitted by: Mohamed Sayed Mohamed

Ahmed Mohamed Makram

Under supervision of: Dr/ Karim Osama

Design of 32-Point Radix-2 FFT architecture Based on Cooley and Tukey algorithm

ABSTRACT

Digital signal processing needs the conversion from time domain to frequency domain. For discrete sequences, this conversion is done through Discrete Fourier Transform but it is numerically inefficient. Thus an efficient algorithm is developed to compute DFT and it is known as Fast Fourier transforms (FFT).In order to fulfill the requirements of executing precise calculations and less power & area consumption, an algorithm with less number of adders and multipliers is used.

INTRODUCTION

Discrete Fourier Transform is a very computationally intensive process that is based on summation a finite series of products of input signal values and trigonometric functions. Its time complexity of the algorithm in O(n2).To increase the performance, several algorithms were proposed which can be implemented in hardware or software. These set of algorithms are known as Fast Fourier Transforms (FFT).The first major FFT algorithm was proposed by Cooley and Tukey which has a time complexity of O(nlogn).

The Fast Fourier transforms (FFT) are the numerically efficient algorithms to compute the Discrete Fourier Transform (DFT). FFT algorithms are based on the concept of divide and conquer approach and it is done by decomposing the computation of DFT into smaller sequences of DFTs. This approach is useful in many areas but calculating it directly from the definition is often very slow to be practical. The FFT is used in so many applications where the frequency-domain representation of a signal has to be processed. In the field of communications the FFT is important because of its use in orthogonal frequency division multiplexing (OFDM) systems.

Cooley-Tukey Algorithm

The Cooley-Tukey FFT is the most universal of all FFT algorithms, because of any factorization of N is possible. FFT algorithms rely on divide and conquer methodology dividing N coefficients points into smaller blocks of different sizes. The first stage computes 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 4 coefficients at this point. This process is repeated until we have one main block, with a final computation of all N coefficients. The computation of the N point DFT via the decimation-in-frequency FFT, as in the decimation-in-time algorithm requires (N/2)log2N complex multiplication and N.log2N complex addition.

The algorithm consists of three main steps:

1. **Bit-reversal:** Where the input samples are re-ordered first before the algorithm starts. The re-ordering is such that; the binary representation of the index of the sample is bit-reversed.

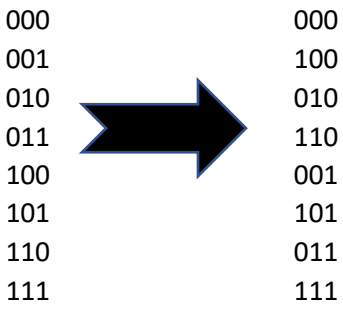


Figure 1: Bit-reversal meathod

2. **Butterfly**: A butterfly is a convenient computational building block with which FFTs are calculated. Using butterflies to draw flow graphs simplifies the diagrams and makes them much easier to read.

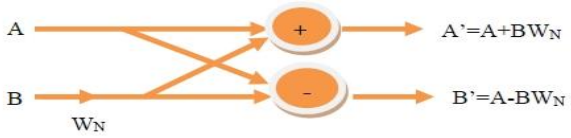


Figure 2: Butterfly Operation

The inputs (A and B) and the outputs (A' and B') of the butterfly are complex numbers containing the data that is being processed. W_N represents a complex sinusoidal value that is applied at each stage of the FFT:

A'= A + BW

B'= A – BW

3. **Twiddle factor:** This is the factor by which the input 2-points are multiplied in the butterfly. They are either stored or calculated on the fly as will be discussed later.

So we can calculate the outputs using these equations:

OUT1= A + BW

OUT2= A – BW

$$(OUT1r + j\,OUT1i) = (Ar + j\,Ai) + [\,(Wr + j\,Wi) \times (Br + j\,Bi)\,]$$

$$(OUT2r + j\,OUT2i) = (Ar + j\,Ai) - [\,(Wr + j\,Wi) \times (Br + j\,Bi)\,]$$

So the output will be: $OUT1\_real = Ar + (Wr*Br - Wi*Bi)$ $OUT1\_image = Ai + (Wr*Bi + Wi*Br)$

$OUT2\_real = Ar - (Wr*Br - Wi*Bi)$ $OUT2\_image = Ai - (Wr*Bi + Wi*Br)$

FFT 32-bit ARCHITECTURE

The radix-2 algorithm is the simplest FFT algorithm with decimation in time. The decimation-in-time (DIT) radix-2 FFT recursively divides a DFT into two half-length DFTs of the even-sequences and odd-sequences of time samples. The outputs of the shorter FFTs are reused to compute many outputs; therefore it is reducing the total computational cost and the delay time.

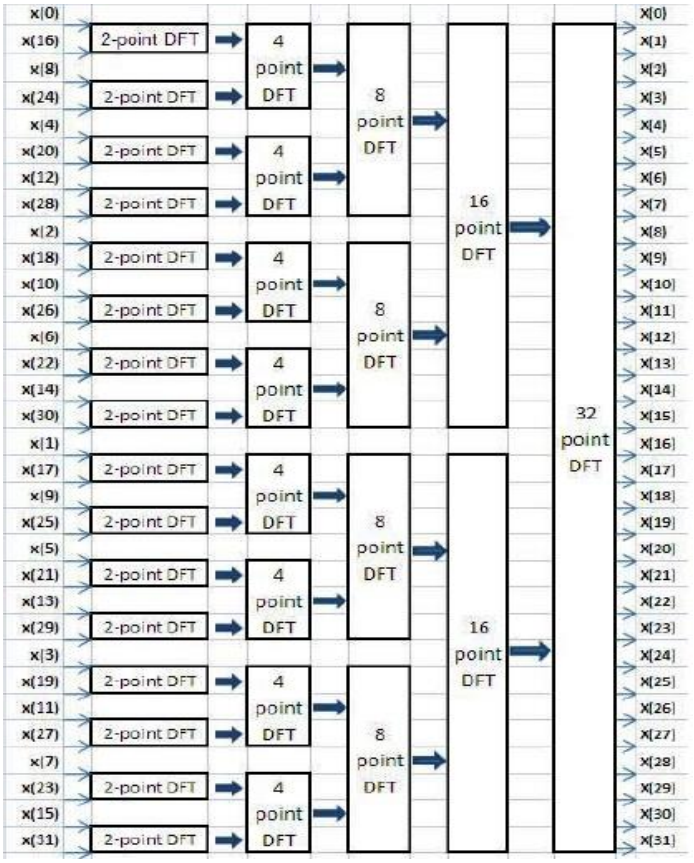


Figure 3: Block Diagram of a 32 Point DIT FFT with Radix-2

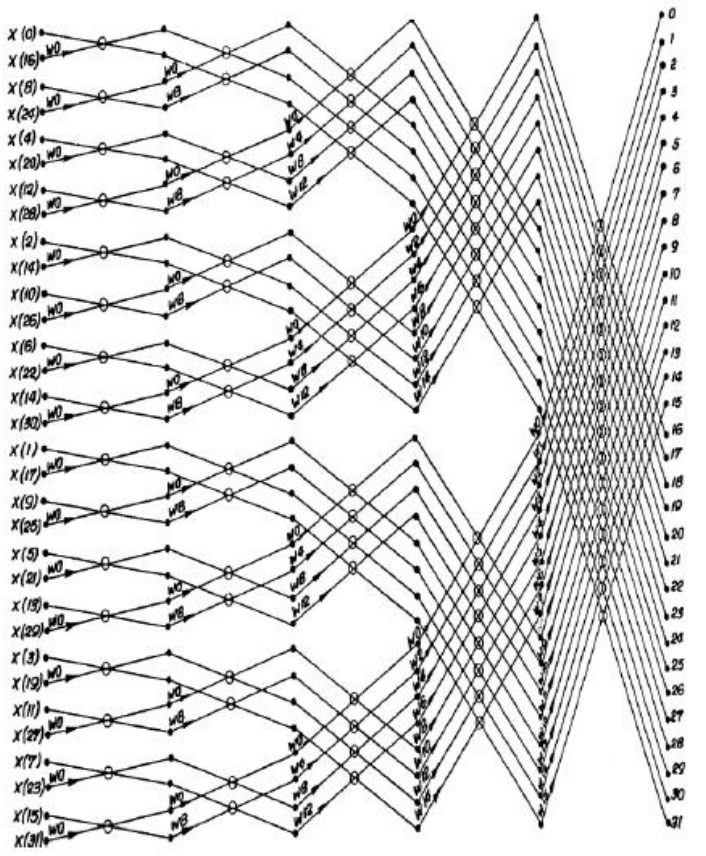


Figure 4: Signal Flow Graph of a 32 Point DIT FFT with Radix-2

Project Assumptions

- Assume a pipelining clock with frequency 20MHz.
- Assume a clock for the MAC units with frequency 100MHz.
- Assume the input samples are 8-bit real and have a signed fixed point representation with 4-bit integer and 4-bit fraction.
- Assume the output samples are 16-bit have a signed fixed point representation with 6-bit integer and 10-bit fraction.

From the assumptions above the input should be in range [-16 : 15.9375] and the output should be in range [-32 : 32] to avoid overflow in the result.

Proposed Design

We have made two different designs for the butterfly block to build the required architecture: one uses only the MAC module as a basic block without using any additional blocks, and the other is used to achieve the minimum resources and to get the full utilization of MAC with take the advantage of its full speed.

MAC module

The multiply–accumulate (MAC) operation is a common step in digital signal processing that computes the product of two numbers then adds that product to an accumulator.

The MAC operation modifies an accumulator as $A \leftarrow A + in1*in2$ each clock cycle, so we need to reset the accumulator register then start to compute the wanted equation.

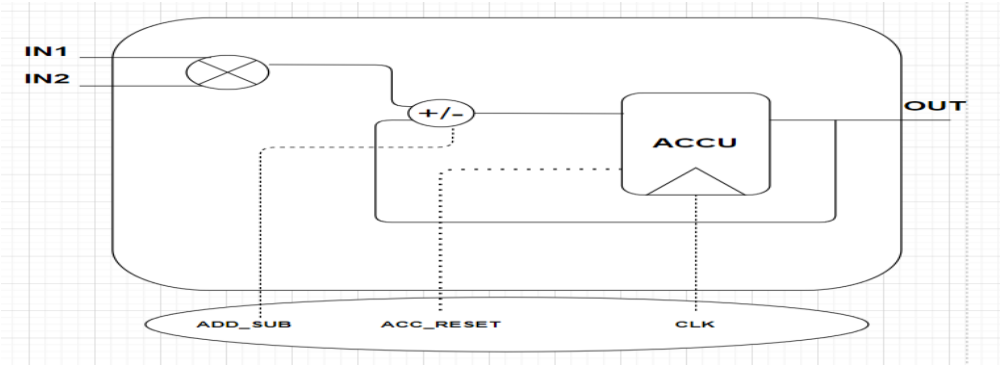


Figure 5: MAC module architecture

Resources:

- #1 16-bit multiplier
- #1 16-bit add/sub block
- #1 16-bit register with reset

Binary Adder-Subtractor

We used in our mac block A Binary Adder-Subtractor which can do both the addition and subtraction operation of binary numbers in one circuit itself.

K=0: addition , K=1: subtraction

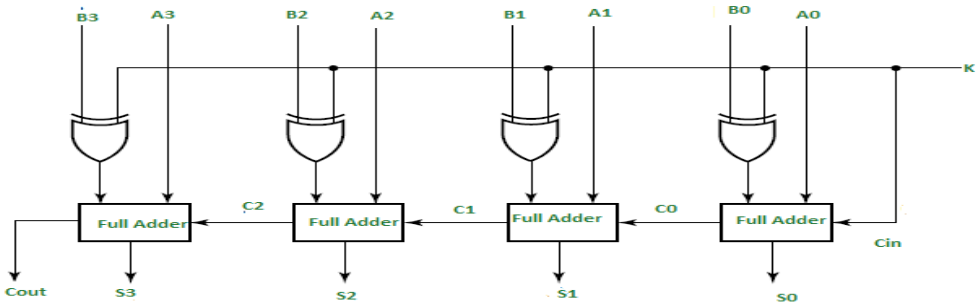


Figure 6: Binary adder-subtractor block

Fixed Point Multiplier Design

We made the fixed point multiplier using one MUL and one XOR to calculate the sign bit then we fixed the corner cases issues using the RTL solutions. To build it using one multiplier instead of four multipliers requires the following procedure:

Example:

For Two 8-bit inputs (4 integer - 4 fraction) and 8-bit output with the same representation:

0011.0100 = 3.2500

X

0010.0001 = 2.0625

_____

0000[0110.1011]0100 = 6.703125

[0110.1011] = 6.6875 *The Output is approximately the same*

For the first stage: We have input 8-bits: (4-bits integer and 4-bits fraction) and we want the output to be 16-bits: (6-bits integer and 10-bits fraction)

So we modified the input enters the first stage to achieve the required number of integer and fraction bits as follow:

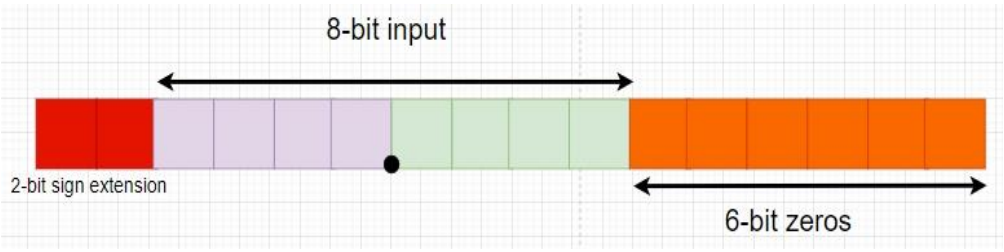


Figure 7: Input Fixed-point shape

We made two-bit sign extension and added six-bit zeros to the right.

We chose this range (6-bits integer and 10-bits fraction) ,as we found that it achieves high accuracy for small numbers and can reach to large numbers as well.

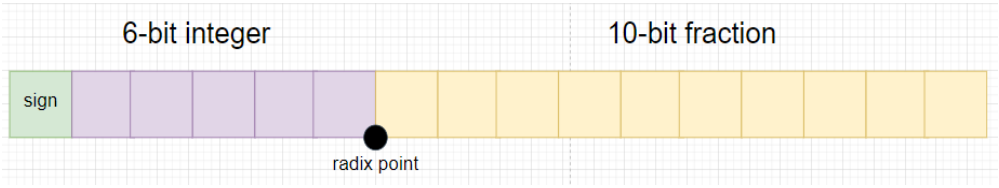


Figure 8: Output Fixed-point shape

First approach architecture

Butterfly design

This design targets using one type of modules to build the butterfly which is the MAC module (we used 2 MAC blocks) and controlling the butterfly with 3-bit counter.

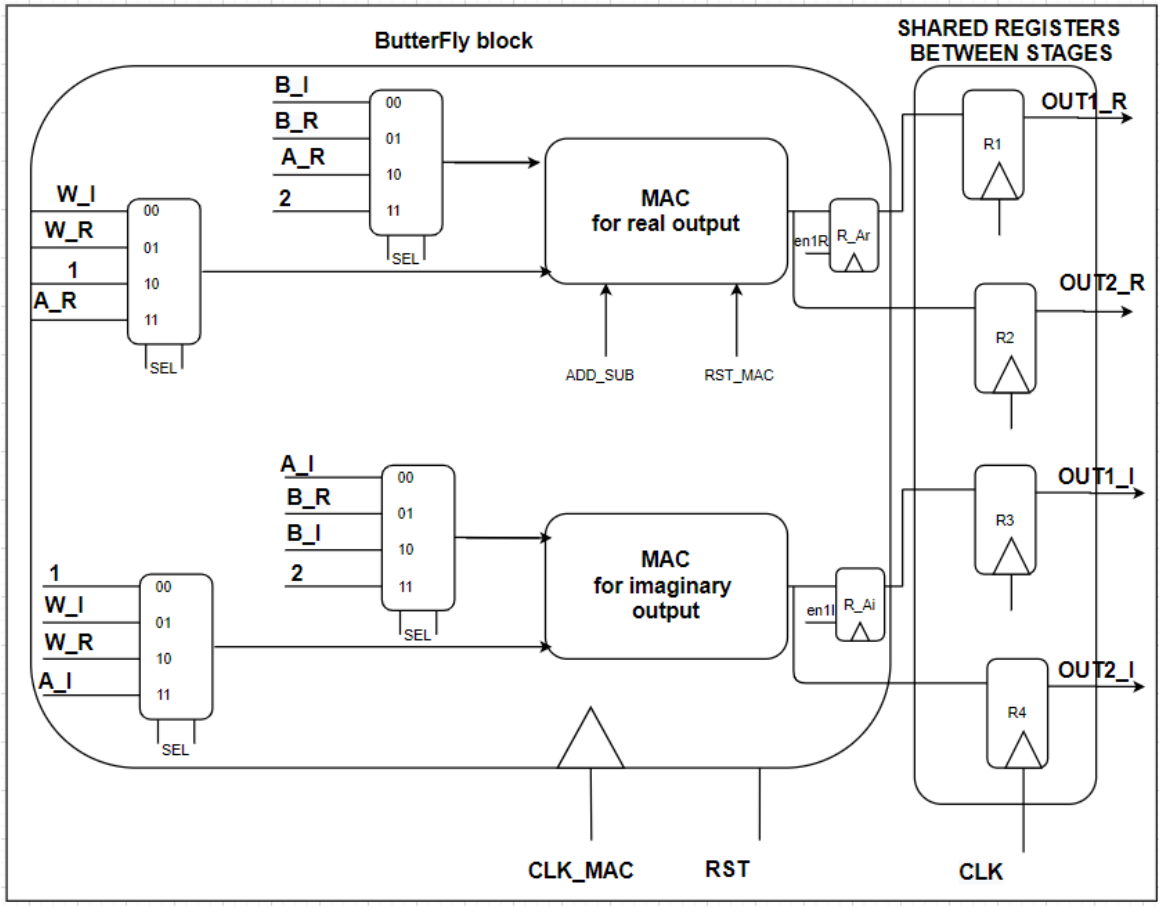


Figure 9: First approach Butterfly architecture

The shown block diagram shows the butterfly followed by the shared registers between stages, we used 2 registers inside the butterfly to save the computed value of A(real and imaginary) for 1 clock cycle , we noticed that **OUT2=2*Ain - OUT1** for real and imaginary components, so we take the advantage of computing "OUT1" in 3 clock cycles then subtract the stored computed value of "OUT1" from **2*Ain** and get "OUT2" in one cycle.

Resources:

- #2 MAC blocks
- #4 4x1 MUX
- #2 16-bit register with Load signal

Controller Design

In this design we used only one 3-bit counter to control all the system

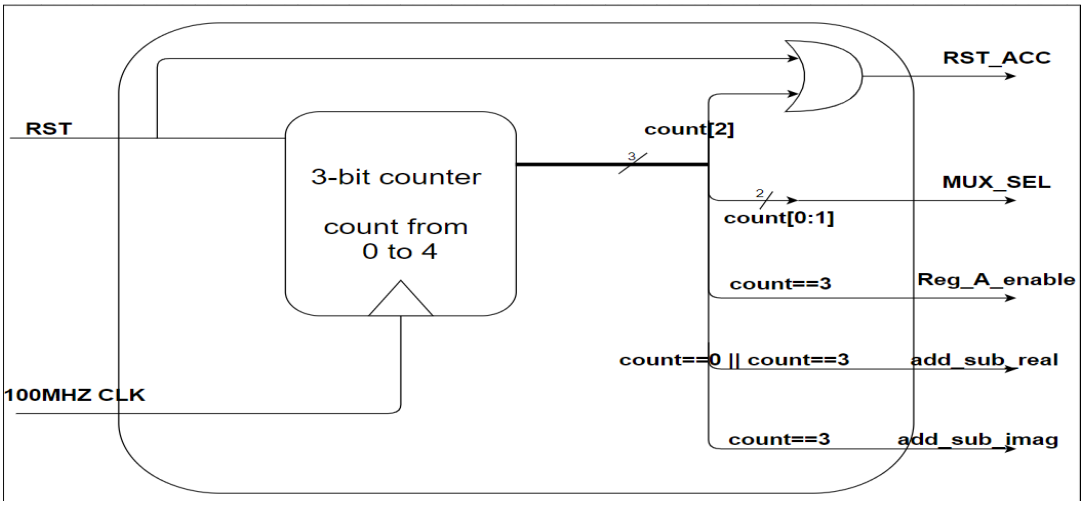


Figure 10: First approach ButterFly controller

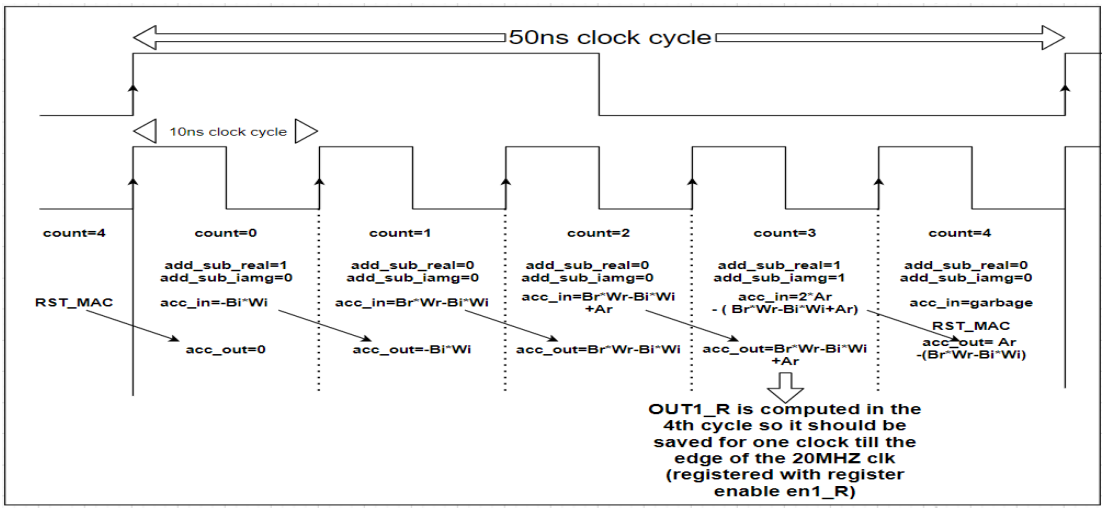


Figure 11: Five cycles of computing the output

Second approach architecture

Butterfly design

This design targets achieving minimum area with highest speed (Full utilization),it can considered to be modification for the first design approach and we completed the design with this approach.

We used only one MAC with one ADD/SUB

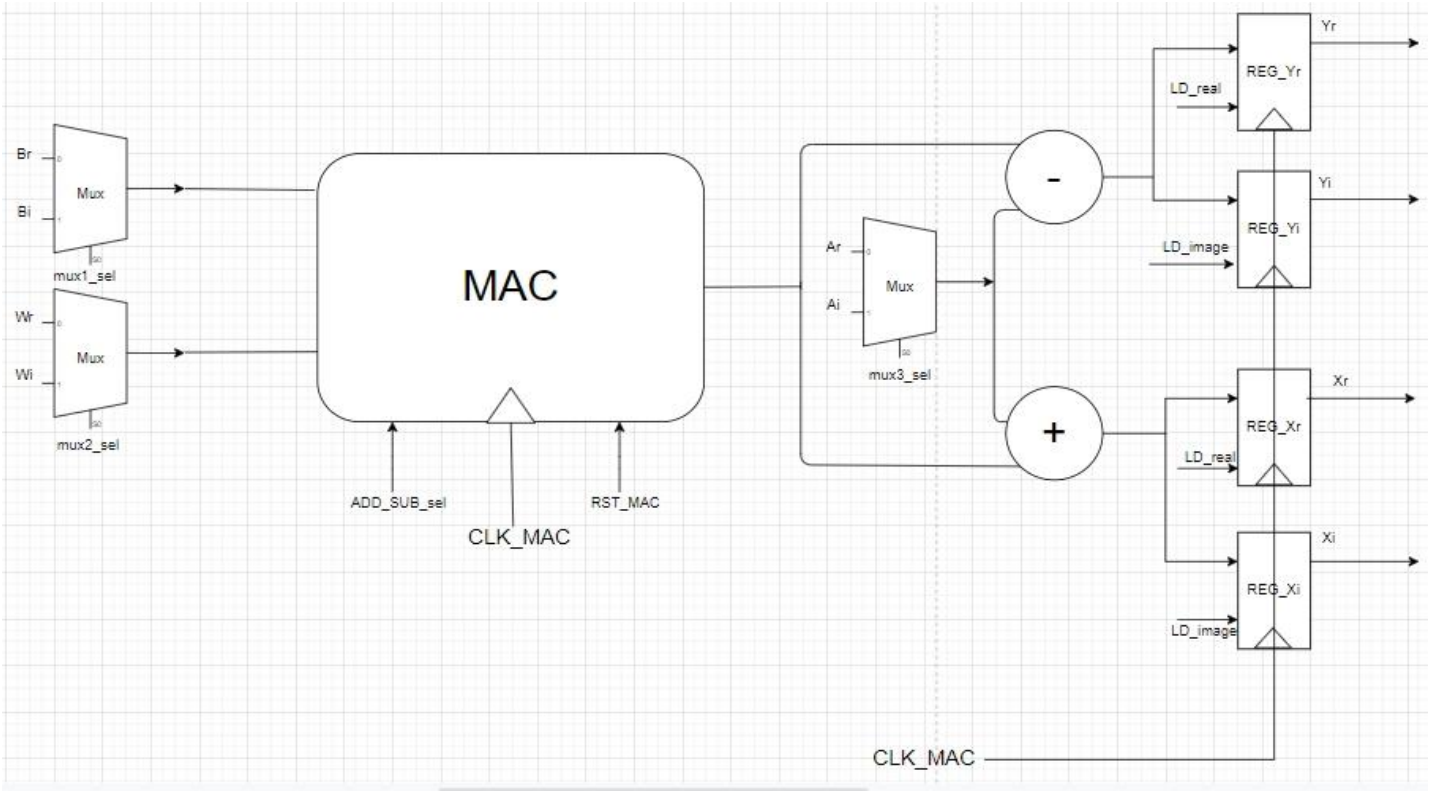


Figure 12: Second approach Butterfly architecture

Resources:

- #1 MAC blocks
- #3 2x1 MUX
- #1 16-bit adder
- #1 16-bit subtractor
- #4 16-bit register with Load signal

Controller Design

We used only one 2-bit counter with 2-bit register to control all the system.

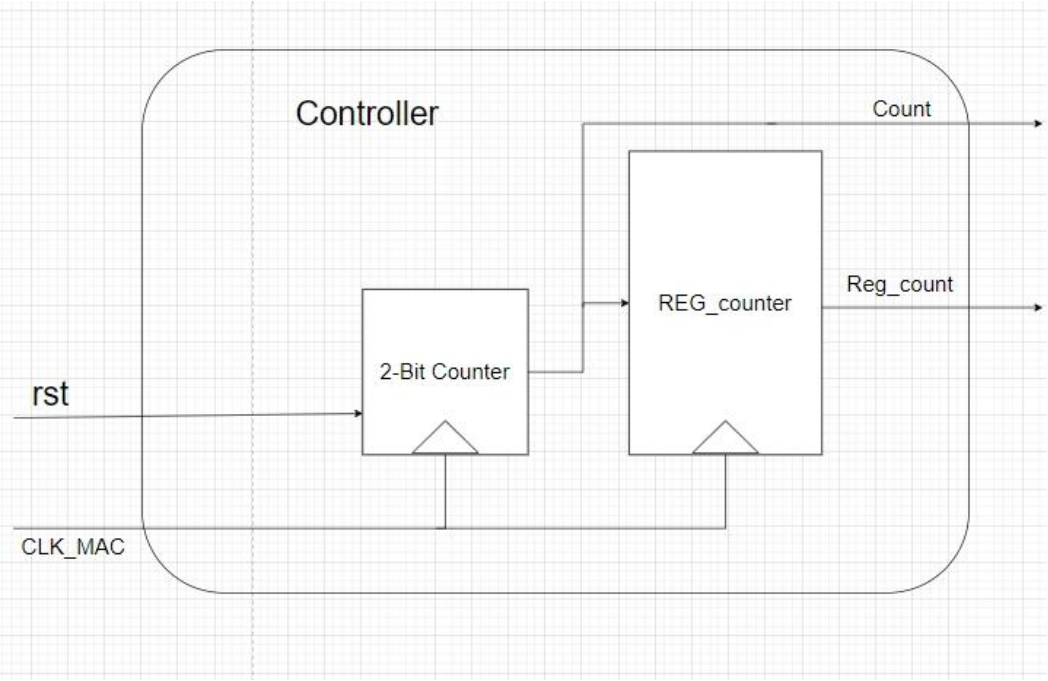


Figure 13: Second approach ButterFly controller

count	sel_W	sel_B	sel_A	ADD_SUB_sel	rst_accumolator	Ld_real	Ld_image
00	0	0	0	0 (ADD)	0	0	1
01	1	1	1	1 (SUB)	1	0	0
10	1	0	0	0 (ADD)	0	1	0
11	0	1	0	0 (ADD)	1	0	0

Table 1: controller output signals

So :

- sel_W= count[0] xor count [1]
- sel_B= count[0]
- sel_A= (count==1)
- rst_accumolator= count[0]
- Ld_real= (count==2)
- Ld_ image = (count==0)

FFT Block Design

Consists of five stages pipelined with ButterFly and intermediate registers between stages.

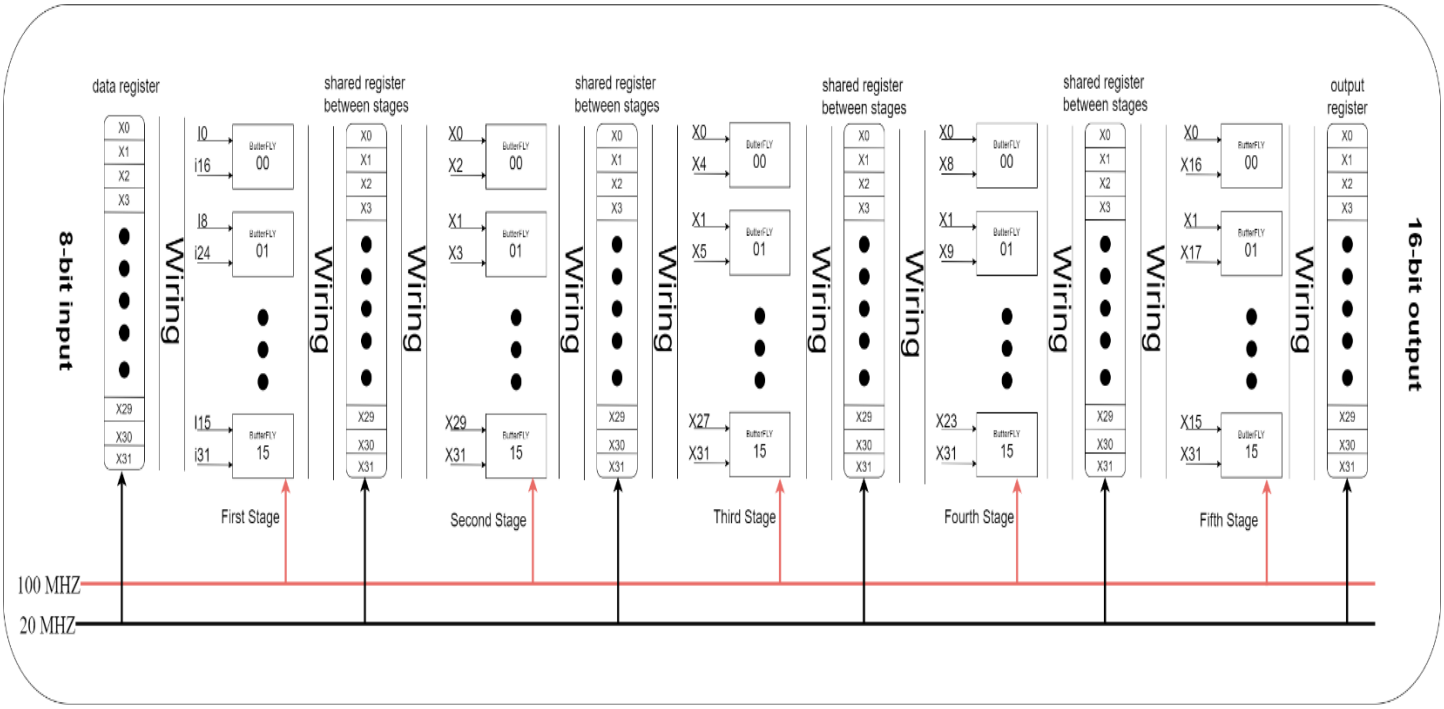


Figure 14:32-point FFT Block Design

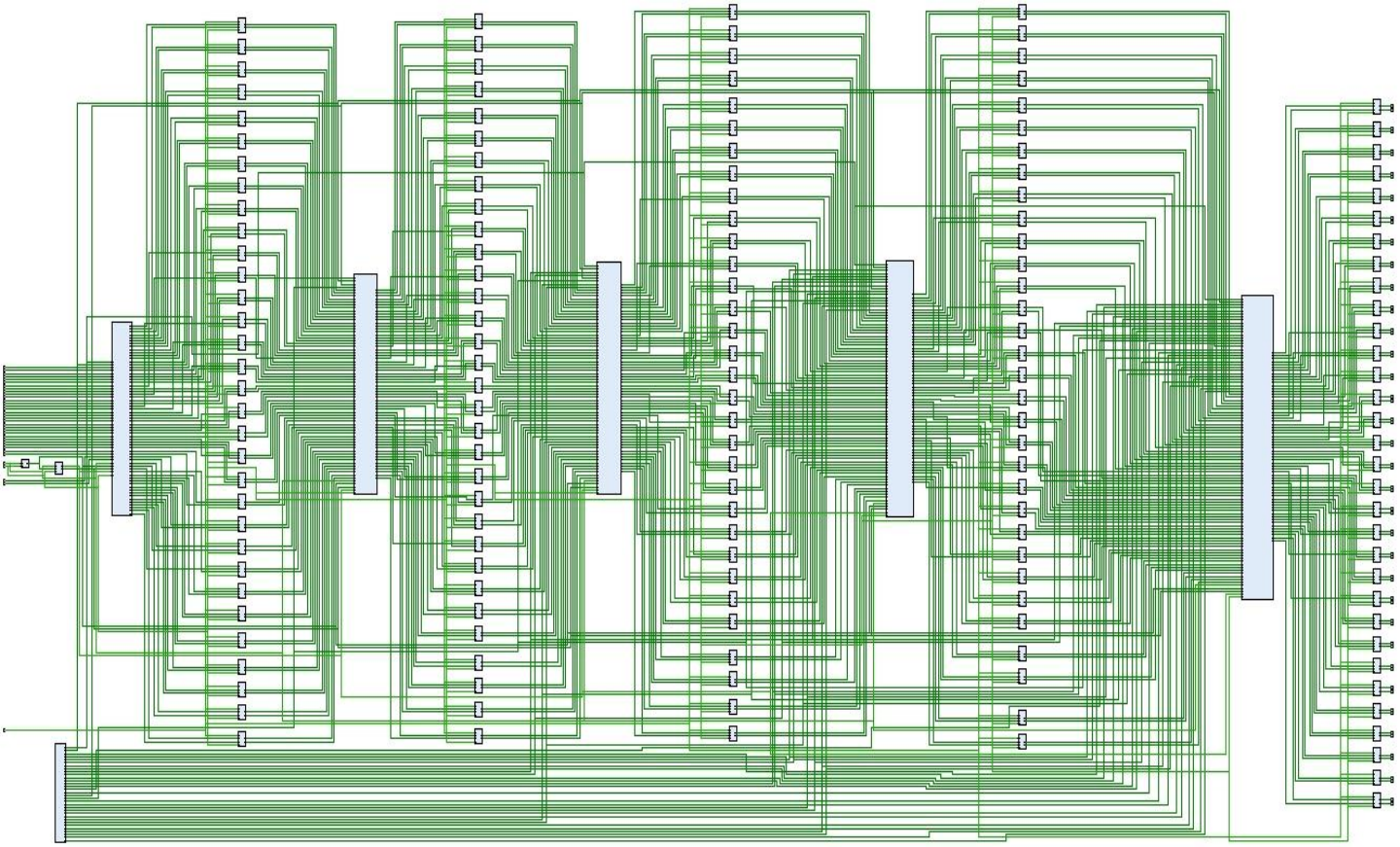


Figure 15: 32-point FFT Schematic using VIVADO

Results

All results obtained using ***ISE Design Suite 14.7*** and ***Vivado 2018.3.***

Resource usage

```
=========================================================================
Advanced HDL Synthesis Report

Macro Statistics
# Multipliers                                          : 80
 16x16-bit multiplier                                  : 80
# Adders/Subtractors                                   : 240
 16-bit adder                                          : 80
 16-bit addsub                                         : 80
 16-bit subtractor                                     : 80
# Counters                                             : 1
 3-bit up counter                                      : 1
# Registers                                            : 10499
 Flip-Flops                                            : 10499
# Multiplexers                                         : 480
 16-bit 2-to-1 multiplexer                             : 480
# Xors                                                 : 80
 1-bit xor2                                            : 80
```

Figure 16:Resource usage using ISE Design SUITE 14.7 Synthesis tool

-Number of multipliers = Number of Butterflies in the system = 5 *16=80 multiplier.

*Number of shared Registers =5*32*2=320 Reg (each 16-bit)*

*Number of Butterflies Registers =80*4=320 Reg (each 16-bit)*

Number of controller Registers = 1 Reg (3-bit)

-So the total number of D Flip-Flop=(320+320)*16+1*3= 10499 Flip-Flops (each 1-bit)

Timing Information

```
Timing Summary:
---------------
Speed Grade: -2

   Minimum period: 6.048ns (Maximum Frequency: 165.349MHz)
   Minimum input arrival time before clock: 5.403ns
   Maximum output required time after clock: 0.580ns
   Maximum combinational path delay: No path found
```

Figure 17: Timing report using ISE Design SUITE 14.7

The worst Propagation delay time in the system is 6.048ns so the system still can work for a frequency =165MHz without any violation.

Pre Synthesis Simulation (Behavioral Simulation)

We made a data set contains Five 32-input data each is 8-bit (1 sign,3 integer , 4 fraction) to test the performance of the pipelining and to calculate an accurate accuracy of the system.

Input Data WaveForm

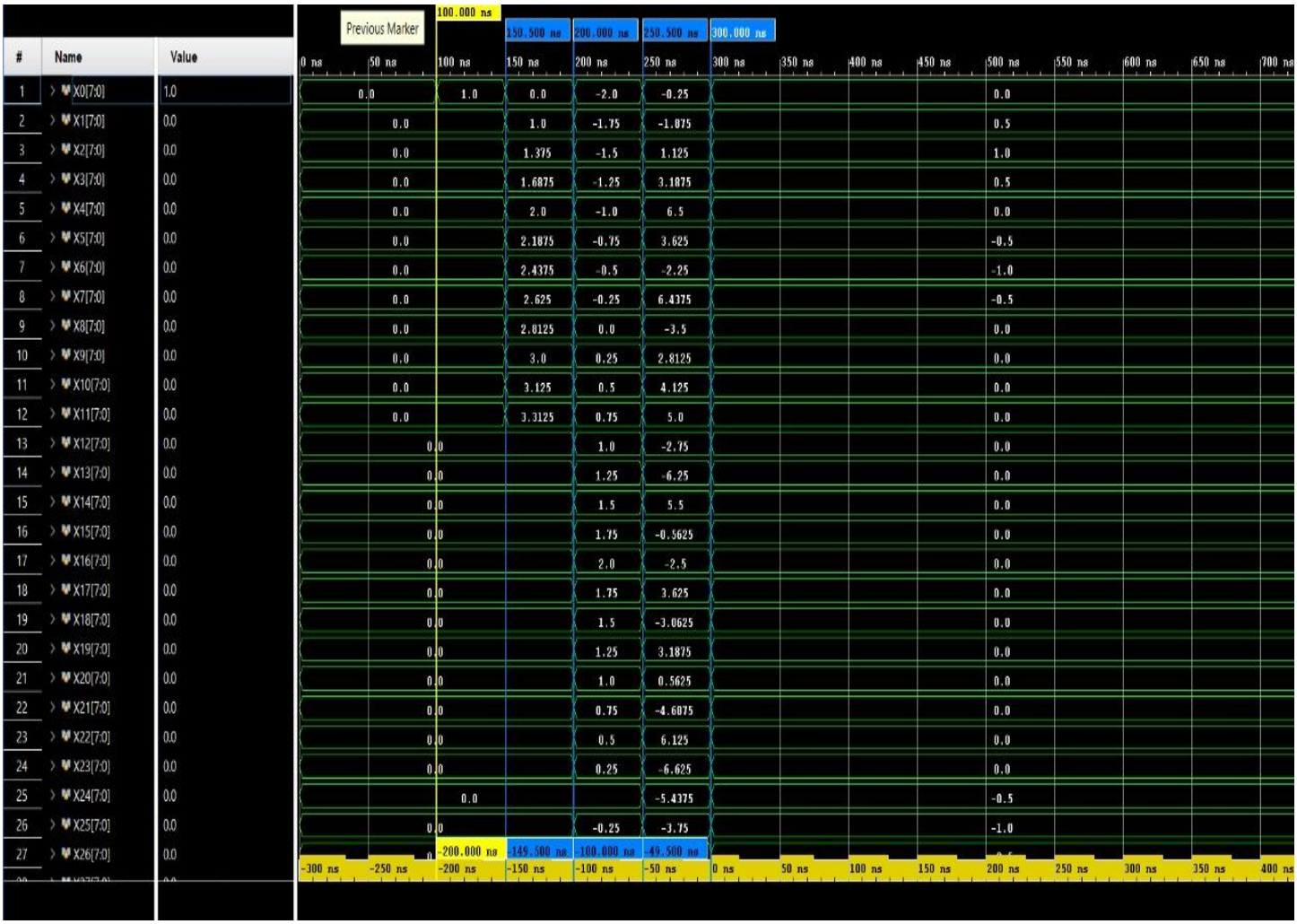


Figure 18: Input data from X0 to X26 waveform

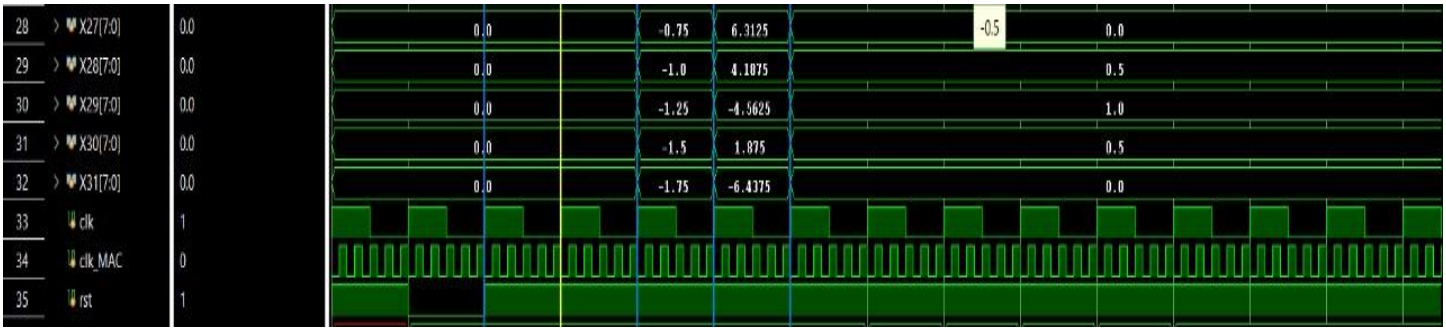


Figure 19: Input data from X27 to X31 waveform

As shown we entered 5 different inputs to the system each clock with frequency 20MHz which is the pipelined frequency.

Output Data WaveForm

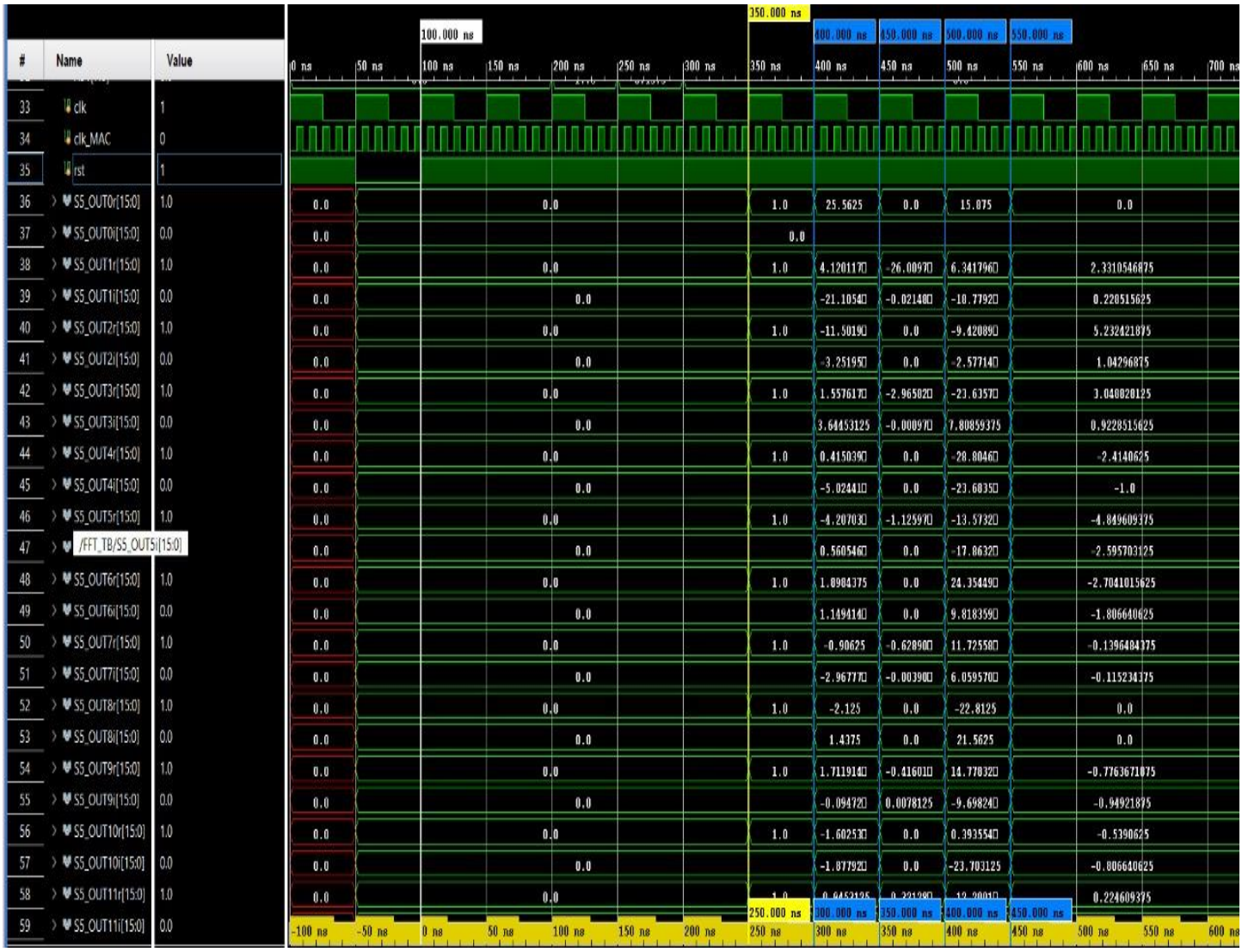


Figure 20: output data from OUT0 to OUT11 waveform

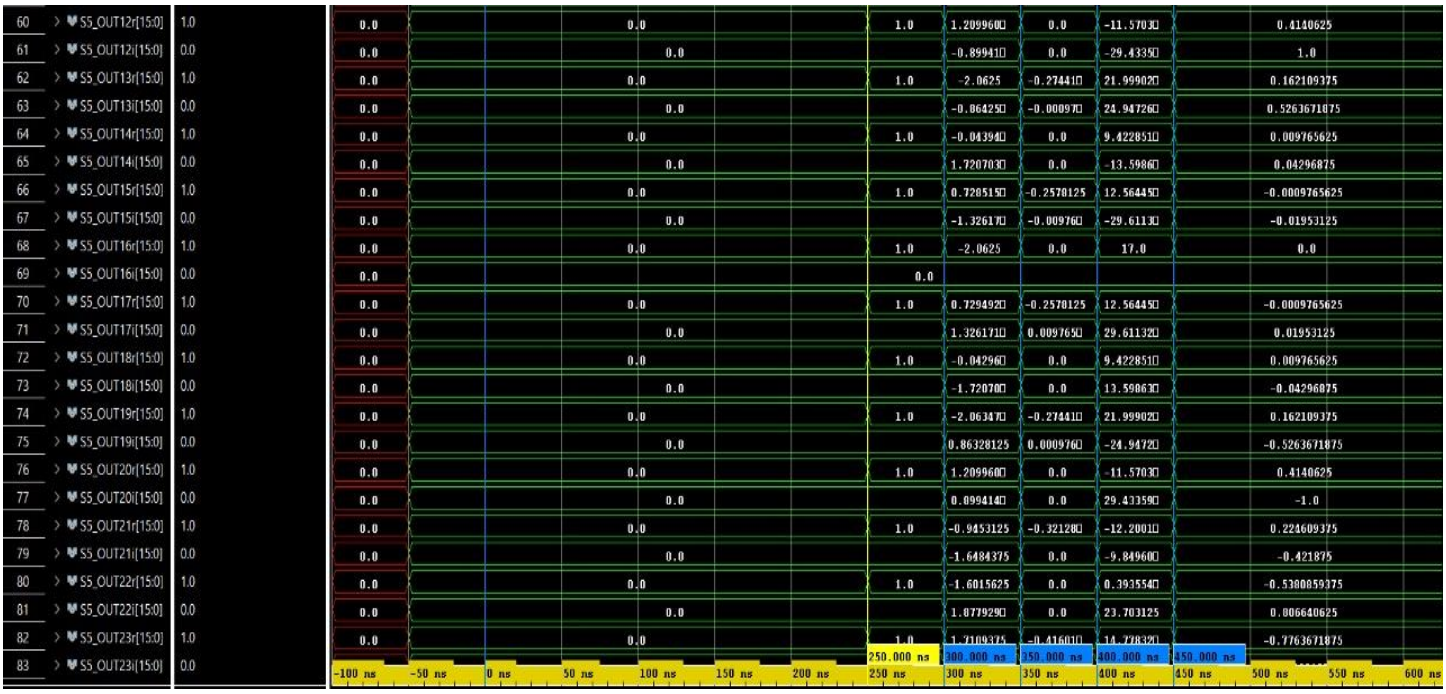


Figure 21: output data from OUT12 to OUT23 waveform

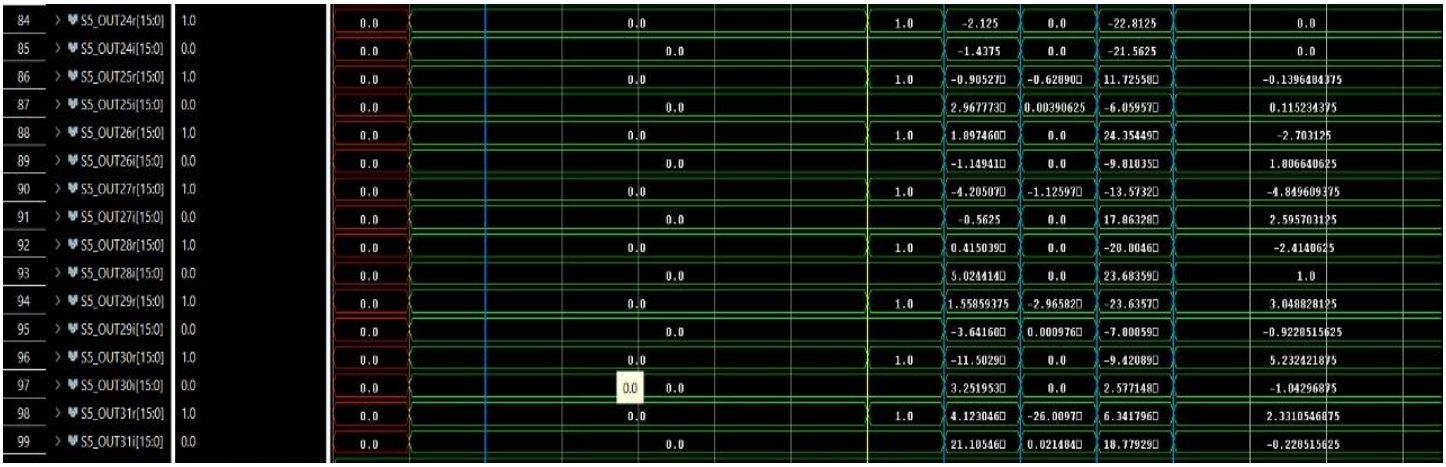


Figure 22: output data from OUT24 to OUT31 waveform

As shown the first output appears after 5 clocks from the first input as the pipelining still empty then the rest outputs appear each clock.

Console Results

We made four tasks to make the testbench more readable for any user.

- initialization TASK : which initialize all the system inputs.
- apply_IN: which apply the stimulus data to the system from the external data-set which we can consider it as emulation for the real DAC output in the full system (i.e. OFDM).
- display_InputData : which displays the input data in decimal to make it easily to read.
- monitor_OUT: which displays the output results in fixed point representation.

Display input data:

```
READING DATA NUMBER:        1

CASE: PULSE INPUT

DISPLAYING INPUT DATA  IN DICEMAL:

X[ 0] =          1.0
X[ 1] =          0.0
X[ 2] =          0.0
X[ 3] =          0.0
X[ 4] =          0.0
X[ 5] =          0.0
X[ 6] =          0.0
X[ 7] =          0.0
X[ 8] =          0.0
X[ 9] =          0.0
X[10] =          0.0
X[11] =          0.0
X[12] =          0.0
X[13] =          0.0
X[14] =          0.0
X[15] =          0.0
X[16] =          0.0
X[17] =          0.0
X[18] =          0.0
X[19] =          0.0
X[20] =          0.0
X[21] =          0.0
X[22] =          0.0
X[23] =          0.0
X[24] =          0.0
X[25] =          0.0
X[26] =          0.0
X[27] =          0.0
X[28] =          0.0
X[29] =          0.0
X[30] =          0.0
X[31] =          0.0
```

```
READING DATA NUMBER:        2

CASE: SQUARE ROOT INPUT

DISPLAYING INPUT DATA  IN DICEMAL:

X[ 0] =     sqrt(0)=0.0
X[ 1] =     sqrt(1)=1.0
X[ 2] =   sqrt(2)=1.414
X[ 3] =   sqrt(3)=1.732
X[ 4] =     sqrt(4)=2.0
X[ 5] =   sqrt(5)=2.236
X[ 6] =   sqrt(6)=2.449
X[ 7] =   sqrt(7)=2.645
X[ 8] =   sqrt(8)=2.828
X[ 9] =     sqrt(9)=3.0
X[10] =   sqrt(10)=3.162
X[11] =   sqrt(11)=3.316
X[12] =          0.0
X[13] =          0.0
X[14] =          0.0
X[15] =          0.0
X[16] =          0.0
X[17] =          0.0
X[18] =          0.0
X[19] =          0.0
X[20] =          0.0
X[21] =          0.0
X[22] =          0.0
X[23] =          0.0
X[24] =          0.0
X[25] =          0.0
X[26] =          0.0
X[27] =          0.0
X[28] =          0.0
X[29] =          0.0
X[30] =          0.0
X[31] =          0.0
```

```
READING DATA NUMBER:        3

CASE: TRIANGLE INPUT

DISPLAYING INPUT DATA  IN DICEMAL:

X[ 0] =         -2.0
X[ 1] =        -1.75
X[ 2] =         -1.5
X[ 3] =        -1.25
X[ 4] =         -1.0
X[ 5] =        -0.75
X[ 6] =         -0.5
X[ 7] =        -0.25
X[ 8] =          0.0
X[ 9] =         0.25
X[10] =          0.5
X[11] =         0.75
X[12] =          1.0
X[13] =         1.25
X[14] =          1.5
X[15] =         1.75
X[16] =          2.0
X[17] =         1.75
X[18] =          1.5
X[19] =         1.25
X[20] =          1.0
X[21] =         0.75
X[22] =          0.5
X[23] =         0.25
X[24] =          0.0
X[25] =        -0.25
X[26] =         -0.5
X[27] =        -0.75
X[28] =         -1.0
X[29] =        -1.25
X[30] =         -1.5
X[31] =        -1.75
```

Figure 23:Displaying input data in Console

Display Output Data:

```
DISPLAYING FFT OUTPUTS NUMBER:        1

CASE: PULSE INPUT

OUT0_real = 000001.0000000000          OUT0_image = 000000.0000000000
OUT1_real = 000001.0000000000          OUT1_image = 000000.0000000000
OUT2_real = 000001.0000000000          OUT2_image = 000000.0000000000
OUT3_real = 000001.0000000000          OUT3_image = 000000.0000000000
OUT4_real = 000001.0000000000          OUT4_image = 000000.0000000000
OUT5_real = 000001.0000000000          OUT5_image = 000000.0000000000
OUT6_real = 000001.0000000000          OUT6_image = 000000.0000000000
OUT7_real = 000001.0000000000          OUT7_image = 000000.0000000000
OUT8_real = 000001.0000000000          OUT8_image = 000000.0000000000
OUT9_real = 000001.0000000000          OUT9_image = 000000.0000000000
OUT10_real = 000001.0000000000         OUT10_image = 000000.0000000000
OUT11_real = 000001.0000000000         OUT11_image = 000000.0000000000
OUT12_real = 000001.0000000000         OUT12_image = 000000.0000000000
OUT13_real = 000001.0000000000         OUT13_image = 000000.0000000000
OUT14_real = 000001.0000000000         OUT14_image = 000000.0000000000
OUT15_real = 000001.0000000000         OUT15_image = 000000.0000000000
OUT16_real = 000001.0000000000         OUT16_image = 000000.0000000000
OUT17_real = 000001.0000000000         OUT17_image = 000000.0000000000
OUT18_real = 000001.0000000000         OUT18_image = 000000.0000000000
OUT19_real = 000001.0000000000         OUT19_image = 000000.0000000000
OUT20_real = 000001.0000000000         OUT20_image = 000000.0000000000
OUT21_real = 000001.0000000000         OUT21_image = 000000.0000000000
OUT22_real = 000001.0000000000         OUT22_image = 000000.0000000000
OUT23_real = 000001.0000000000         OUT23_image = 000000.0000000000
OUT24_real = 000001.0000000000         OUT24_image = 000000.0000000000
OUT25_real = 000001.0000000000         OUT25_image = 000000.0000000000
OUT26_real = 000001.0000000000         OUT26_image = 000000.0000000000
OUT27_real = 000001.0000000000         OUT27_image = 000000.0000000000
OUT28_real = 000001.0000000000         OUT28_image = 000000.0000000000
OUT29_real = 000001.0000000000         OUT29_image = 000000.0000000000
OUT30_real = 000001.0000000000         OUT30_image = 000000.0000000000
OUT31_real = 000001.0000000000         OUT31_image = 000000.0000000000
```

```
DISPLAYING FFT OUTPUTS NUMBER:        2

CASE: SQUARE ROOT INPUT

OUT0_real = 011001.1001000000          OUT0_image = 000000.0000000000
OUT1_real = 000100.0001111011          OUT1_image = 101010.1110010100
OUT2_real = 110100.0111111110          OUT2_image = 111100.1011111110
OUT3_real = 000001.1000111011          OUT3_image = 000011.1010010100
OUT4_real = 000000.0110101001          OUT4_image = 111010.1111100111
OUT5_real = 111011.1100101100          OUT5_image = 000000.1000111110
OUT6_real = 000001.1110011000          OUT6_image = 000001.0010011001
OUT7_real = 111111.0001100000          OUT7_image = 111101.0000100001
OUT8_real = 111101.1110000000          OUT8_image = 000001.0111000000
OUT9_real = 000001.1011011001          OUT9_image = 111111.1110011111
OUT10_real = 111110.0110010111         OUT10_image = 111110.0001111101
OUT11_real = 111111.0000111000         OUT11_image = 000001.1010011000
OUT12_real = 000001.0011010111         OUT12_image = 111111.0001100111
OUT13_real = 111101.1111000000         OUT13_image = 111111.0010001011
OUT14_real = 111111.1110110011         OUT14_image = 000001.1011100010
OUT15_real = 000000.1011101010         OUT15_image = 111110.1010110010
OUT16_real = 111101.1111000000         OUT16_image = 000000.0000000000
OUT17_real = 000000.1011101011         OUT17_image = 000001.0101001110
OUT18_real = 111111.1110101100         OUT18_image = 111110.0100011110
OUT19_real = 111101.1110111111         OUT19_image = 000000.1101110100
OUT20_real = 000001.0011010111         OUT20_image = 000000.1110011001
OUT21_real = 111111.0000111000         OUT21_image = 111110.0101101000
OUT22_real = 111110.0110011000         OUT22_image = 000001.1110000011
OUT23_real = 000001.1011011000         OUT23_image = 000000.0001100001
OUT24_real = 111101.1110000000         OUT24_image = 111110.1001000000
OUT25_real = 111111.0001100001         OUT25_image = 000010.1111011111
OUT26_real = 000001.1110010111         OUT26_image = 111110.1101100111
OUT27_real = 111011.1100101110         OUT27_image = 111111.0111000000
OUT28_real = 000000.0110101001         OUT28_image = 000101.0000011001
OUT29_real = 000001.1000111100         OUT29_image = 111100.0101101111
OUT30_real = 110100.0111111101         OUT30_image = 000011.0100000010
OUT31_real = 000100.0001111110         OUT31_image = 010101.0001101100
```

```
DISPLAYING FFT OUTPUTS NUMBER:       3

CASE: TRIANGLE INPUT

OUT0_real = 000000.0000000000          OUT0_image = 000000.0000000000
OUT1_real = 100101.1111110110          OUT1_image = 111111.1111101010
OUT2_real = 000000.0000000000          OUT2_image = 000000.0000000000
OUT3_real = 111101.0000100011          OUT3_image = 111111.1111111111
OUT4_real = 000000.0000000000          OUT4_image = 000000.0000000000
OUT5_real = 111110.1101111111          OUT5_image = 000000.0000000000
OUT6_real = 000000.0000000000          OUT6_image = 000000.0000000000
OUT7_real = 111111.0101111100          OUT7_image = 111111.1111111100
OUT8_real = 000000.0000000000          OUT8_image = 000000.0000000000
OUT9_real = 111111.1001010110          OUT9_image = 000000.0000001000
OUT10_real = 000000.0000000000         OUT10_image = 000000.0000000000
OUT11_real = 111111.1010110111         OUT11_image = 000000.0000000000
OUT12_real = 000000.0000000000         OUT12_image = 000000.0000000000
OUT13_real = 111111.1011100111         OUT13_image = 111111.1111111111
OUT14_real = 000000.0000000000         OUT14_image = 000000.0000000000
OUT15_real = 111111.1011111000         OUT15_image = 111111.1111110110
OUT16_real = 000000.0000000000         OUT16_image = 000000.0000000000
OUT17_real = 111111.1011111000         OUT17_image = 000000.0000001010
OUT18_real = 000000.0000000000         OUT18_image = 000000.0000000000
OUT19_real = 111111.1011100111         OUT19_image = 000000.0000000001
OUT20_real = 000000.0000000000         OUT20_image = 000000.0000000000
OUT21_real = 111111.1010110111         OUT21_image = 000000.0000000000
OUT22_real = 000000.0000000000         OUT22_image = 000000.0000000000
OUT23_real = 111111.1001010110         OUT23_image = 111111.1111111000
OUT24_real = 000000.0000000000         OUT24_image = 000000.0000000000
OUT25_real = 111111.0101111100         OUT25_image = 000000.0000000100
OUT26_real = 000000.0000000000         OUT26_image = 000000.0000000000
OUT27_real = 111110.1101111111         OUT27_image = 000000.0000000000
OUT28_real = 000000.0000000000         OUT28_image = 000000.0000000000
OUT29_real = 111101.0000100011         OUT29_image = 000000.0000000001
OUT30_real = 000000.0000000000         OUT30_image = 000000.0000000000
OUT31_real = 100101.1111110110         OUT31_image = 000000.0000010110
```

Figure 24:Displaying output data in console

Post Synthesis Simulation (Timing Simulation)

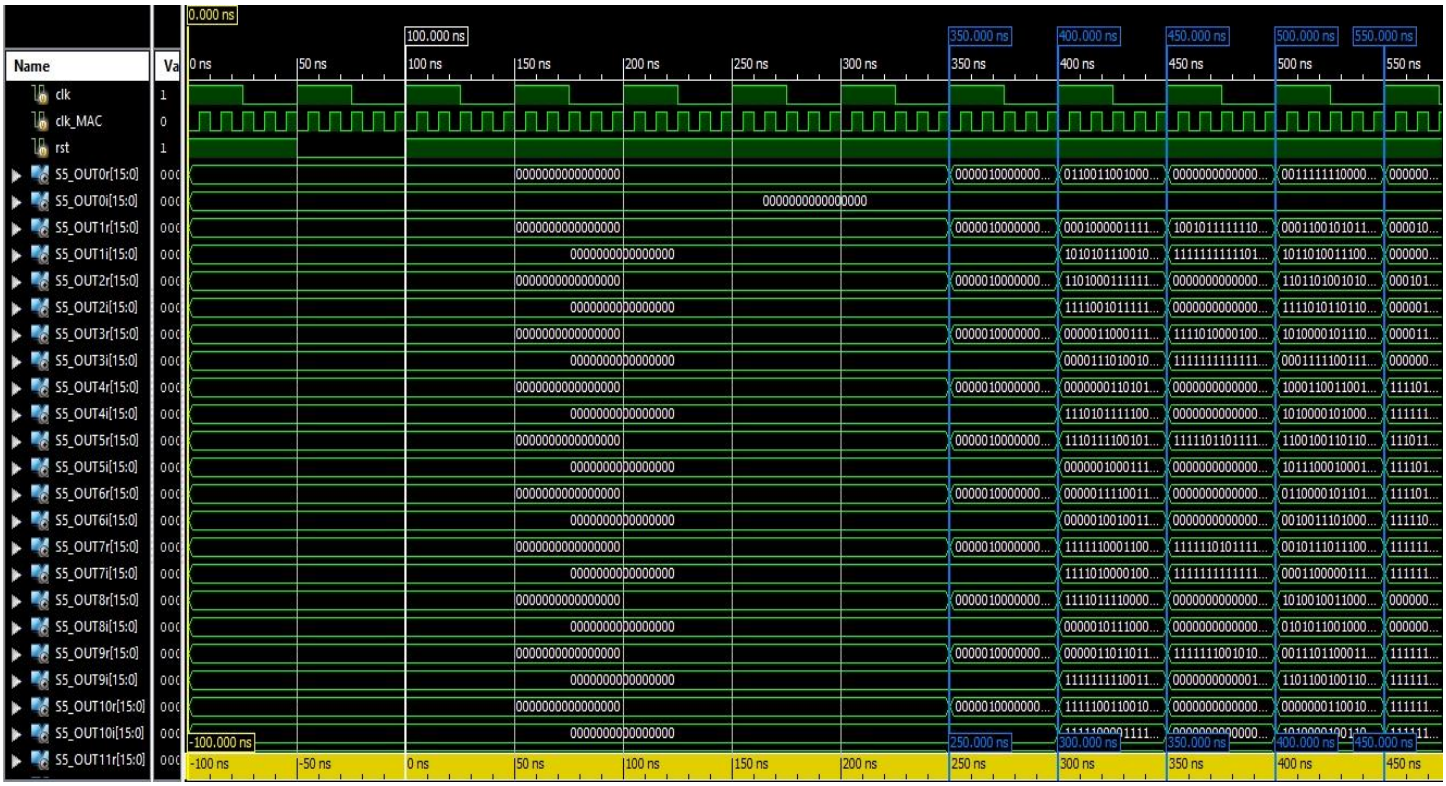


Figure 25:Output Waveform (Timing simulation)

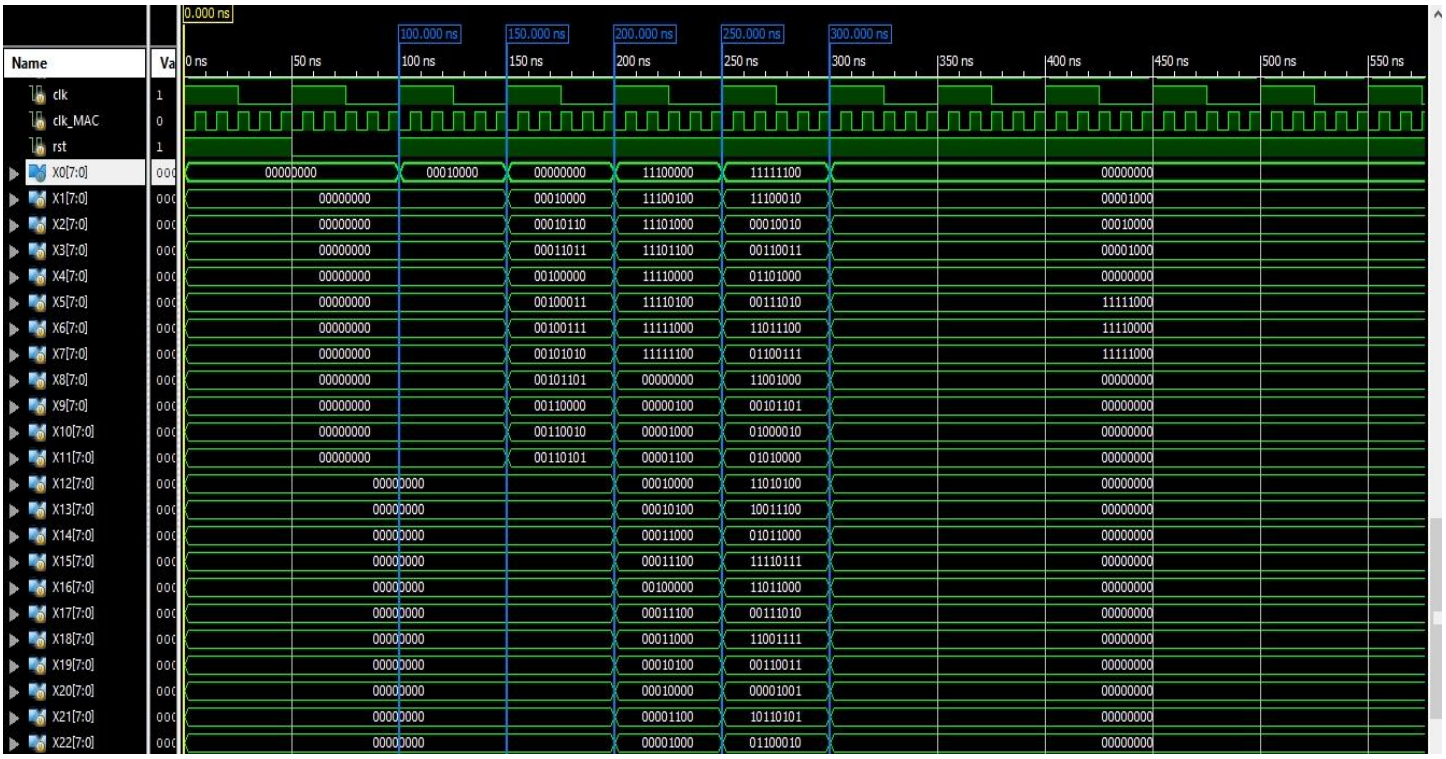


Figure 26:Input Waveform (Timing Simulation)

As shown we got the same result as in the Behavioral Simulation which means we don't have any timing violations.

Power Estimation

We used VIVADO Power Estimator to get the design power dissipated from the synthesized netlist.

Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

Total On-Chip Power:	**0.458 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**25.6°C**
Thermal Margin:	59.4°C (40.9 W)
Effective ϑJA:	1.4°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

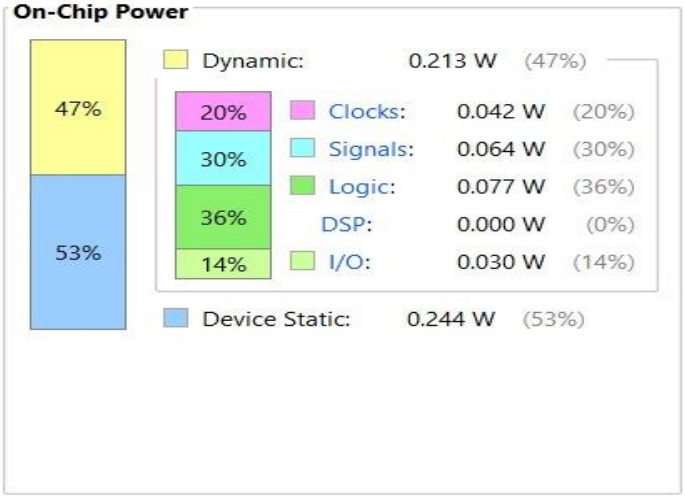


Figure 27:Total On-Chip Power

As shown in figure, the total On-Chip power is 0.458W and we can use power reduction technique to reduce it more.

Verification Part

We used python for generating test vectors using the Built-In *NumPy* function “np.FFT” and converting the generated input to binary fixed point bits (1 sign , 3 integer and 4 fraction bits) using python, then this output is used as RTL input, finally we get the binary bits output of the RTL code to python then convert it to decimal numbers to be easily compared with the python function’s output as in figure26 .

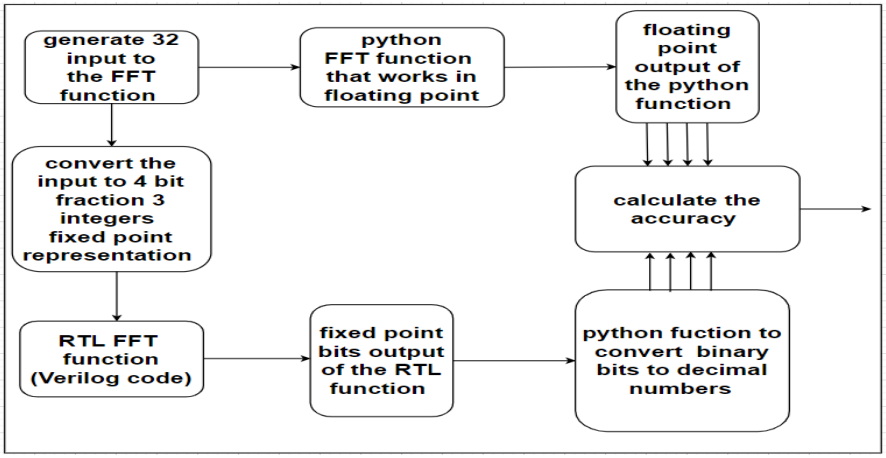


Figure 28: Python flow

Error Calculation

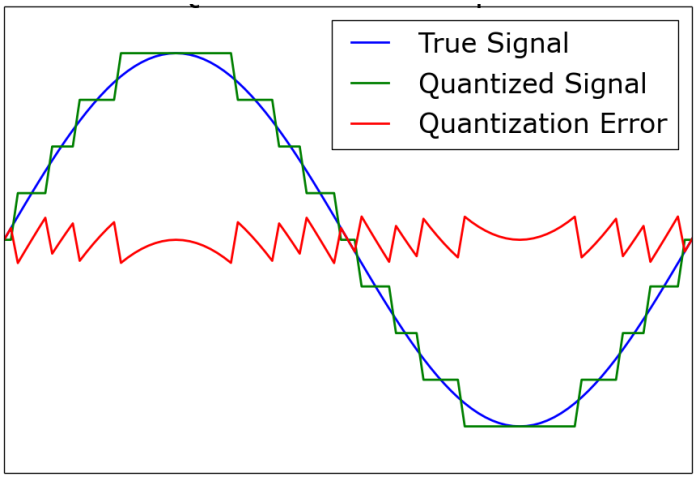
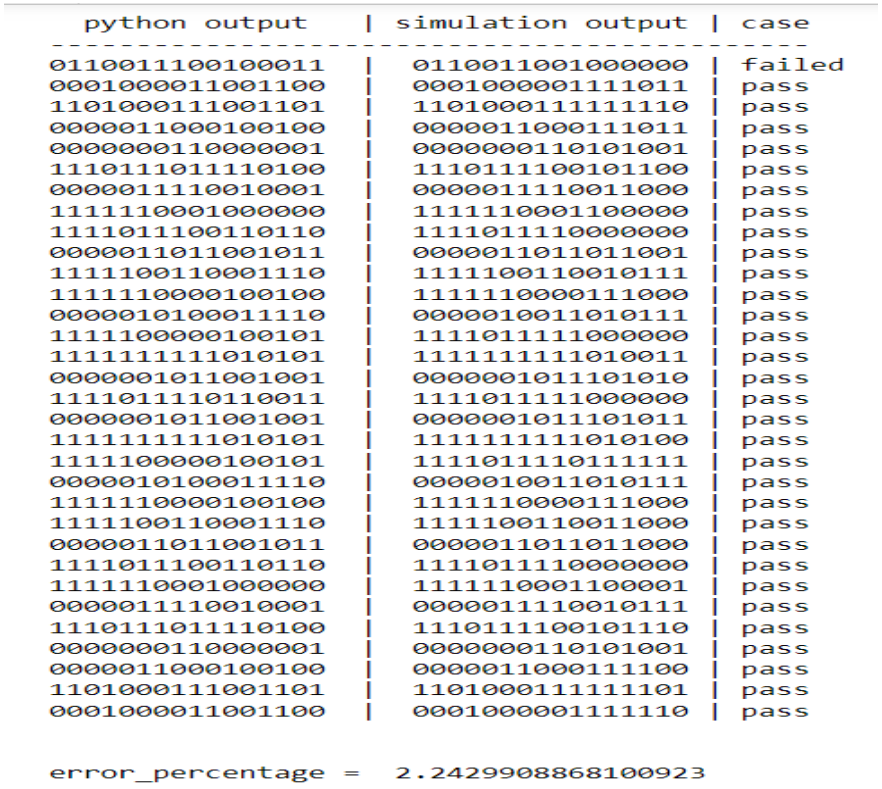


Figure 29: Quantization error waveform

$$\text{Total Error } = \sum_{i=0}^{31} abs(X[i]_{accurate} - X[i]_{RTL})$$

$$\text{Rationalized Error } = \frac{total\ error}{max - min} = \frac{total\ error}{64}$$



python output	simulation output	case
0110011100100011	0110011001000000	failed
0001000011001100	0001000001111011	pass
1101000111001101	1101000111111110	pass
0000011000100100	0000011000111011	pass
0000000110000001	0000000110101001	pass
1110111011110100	1110111100101100	pass
0000011110010001	0000011110011000	pass
1111110001000000	1111110001100000	pass
1111011100110110	1111011110000000	pass
0000011011001011	0000011011011001	pass
1111100110001110	1111100110010111	pass
1111110000100100	1111110000111000	pass
0000010100011110	0000010011010111	pass
1111100000100101	1111011111000000	pass
1111111111010101	1111111111010011	pass
0000001011001001	0000001011101010	pass
1111011110110011	1111011111000000	pass
0000001011001001	0000001011101011	pass
1111111111010101	1111111111010100	pass
1111100000100101	1111011110111111	pass
0000010100011110	0000010011010111	pass
1111110000100100	1111110000111000	pass
1111100110001110	1111100110011000	pass
0000011011001011	0000011011011000	pass
1111011100110110	1111011110000000	pass
1111110001000000	1111110001100001	pass
0000011110010001	0000011110010111	pass
1110111011110100	1110111100101110	pass
0000000110000001	0000000110101001	pass
0000011000100100	0000011000111100	pass
1101000111001101	1101000111111101	pass
0001000011001100	0001000001111110	pass

error_percentage = 2.2429908868100923

Figure 30: Error calculation

We decide the case failed or pass by comparing the absolute of subtraction by threshold value $abs(X[i]_{accurate} - X[i]_{RTL}) < threshold$,we chose the threshold as 0.1 which is in the range of the error values.

So finally our accuracy of the Fixed-Point FFT is **97.757%** and we can increase it if we increased the size of inputs and outputs, but there's a tradeoff between the accuracy and the area.

Conclusion

We made 2 designs for the Cooley Tukey FFT algorithm, one design with only one MAC block and 2 adders in each butterfly and the other design is with 2 MAC blocks, the 2 designs are fully utilized. We implemented the designs using Verilog RTL then we verified that the block is working well using functional and timing simulation. Then we synthesized the RTL using vertix7 FPGA library and verified that the block is also working well after synthesis.

The verification is done mainly using python scripts that give the accuracy and the quantization errors in the outputs due to the fixed point representation.

Appendix

All test codes using Python are uploaded on this link:

https://github.com/mak-rram/python_FFT_project