Algorithm for Resource-Optimized Design of any N-point FFT-Computation

Prasanna Kansakar[1], Sandesh Ghimire[2], Bikash Poudel[1]

[1] *Department of Electronics and Computer Engineering, Thapathali Campus*

[2] *Engineer, Nepal Electricity Authority*

Corresponding Email: prasanna.kansakar@gmail.com

Abstract: This paper presents an algorithm for the design of an FFT computation core, based on the Radix-2 Cooley Tukey algorithm, by reusing a single cross-computation block. The cross computation block consists of a network of adders/subtractors and multipliers which together makeup a single unit of Cooley Tukey butterfly network. The twiddle factor for each cross-computation is generated using a CORDIC processor and supplied to the cross-computation block. Since the design reuses a single cross-computation block, it puts forward a new method with a considerable minimization in the hardware resources for FFT computation as well as simplifies the overall architecture of the FFT core. Despite the fact that same cross-computation hardware block is repetitively used throughout the calculation, the performance time of the design stays within tolerable limits. Using only one cross-computation block also allows the design to be scaled to any N-point FFT without any change in the design description. For example, this algorithm can be used for 8-point or for 1024-point FFT simply by changing the design parameter N. The results from the simulation of a 16-point FFT module, designed using this algorithm, shows that the FFT – coefficients are precise up to 4 digits.

Keywords: N-point FFT; Cooley Tukey FFT; Radix-2 DIT-FFT; CORDIC processor

1. Introduction

The Cooley Tukey Radix-2 Decimation in Time - Fast Fourier Transform (DIT-FFT) is a popular algorithm for implementation of FFT computation. But, one of the drawbacks of this algorithm is that it requires a large quantity of components such as adders, subtractors and multiplier as the number of point increases, which results in an increase in size of hardware.

There are many FFT computation modules available but, they are designed such that the number of points taken in the FFT computation is fixed. This means that in order to go from an 8-point FFT core to a 16-point FFT core, either the design has to be changed completely or significant additions to the design description have to be made.

This paper presents an algorithm that uses the inherent repetitiveness of the Cooley Tukey Radix-2 DIT-FFT technique to implement FFT computation by reusing a single cross-computation block. The reusing of the single block also permits this algorithm to be scaled to any point FFT. The resulting design is thus a resource-optimized and simplest architecture FFT computation module that can be scaled to calculate any N-point FFT.

One obstacle in trying to write a common algorithm for any N-point FFT is calculation of twiddle factors for FFT computation. Usually, N is fixed and a look up table is generated consisting of all twiddle factors for the given value of N, which is later referenced during calculation. To apply the algorithm to any N, this method becomes inefficient because the size of the look up table is indeterminate when value of N is not defined. One of the possible solutions to this problem is to select a very high value of N and generate look up table. However, that would result in wastage of memory if the value of N is considerably low, as is the case in most implementations. Therefore, in this algorithm CORDIC processor is used to calculate all the twiddle factors right at the time they are required.

In section 2, Radix-2 DIT-FFT Cooley Tukey algorithm and CORDIC algorithm have been explained. Section 3 contains detailed explanation of the computation algorithm followed by experiment for testing, result and conclusion in sections 4, 5 and 6 respectively.

2. Background

2.1 Radix-2 decimation in time FFT (DIT-FFT)

In this paper, the Radix-2 Cooley Tukey (Wikipedia, 2014) Decimation in Time (DIT-FFT) algorithm is used for FFT calculation. The DIT-FFT algorithm is an efficient method of calculating Discrete Fourier Transform (DFT) (Oppenheim, Schafer, & Buck, 2012) of a series of non-periodic signals. DFT is defined as

$$X(k) = \sum_{n=0}^{N-1} x[n] e^{-jkn2\pi/N}$$

$$= \sum_{n=0}^{N-1} x[n] w_N^{kn}$$

where,

$$w_N = e^{-j2\pi/N}$$

Separating x$[n]$ into even and odd numbered points,

$$X(k) = \sum_{n=even} x[n] w_N^{kn} + \sum_{n=odd} x[n] w_N^{kn}$$

Substituting $n = 2m$ for even segment and $n = 2m + 1$ for odd segment,

$$X(k) = \sum_{m=0}^{\frac{N}{2}-1} x[2m] w_N^{2km} + \sum_{m=0}^{\frac{N}{2}-1} x[2m-1] w_N^{k(2m+1)}$$

$$= \sum_{m=0}^{\frac{N}{2}-1} x[2m] w_{N/2}^{km} + w_N^k \sum_{m=0}^{\frac{N}{2}-1} x[2m+1] w_{N/2}^{km}$$

$$= G(k) + w_N^k H(k) \cdots\cdots\cdots\cdots (1)$$

where,

$$G(k) = \sum_{m=0}^{\frac{N}{2}-1} x[2m] w_{N/2}^{km}$$

is the DFT of even numbered points of sequence $x[n]$, and

$$H(k) = \sum_{m=0}^{\frac{N}{2}-1} x[2m+1] w_{N/2}^{km}$$

is the DFT of odd numbered points of sequence $x[n]$.

Thus, N point FFT has been simplified to two $\frac{N}{2}$-point FFTs. By continuing in this fashion, a butterfly structure of $\left(\frac{N}{2}\log_2 N\right)$ 2-point FFTs is obtained. Figure 1 shows the butterfly structure for DIT-FFT computation of 8 input points.

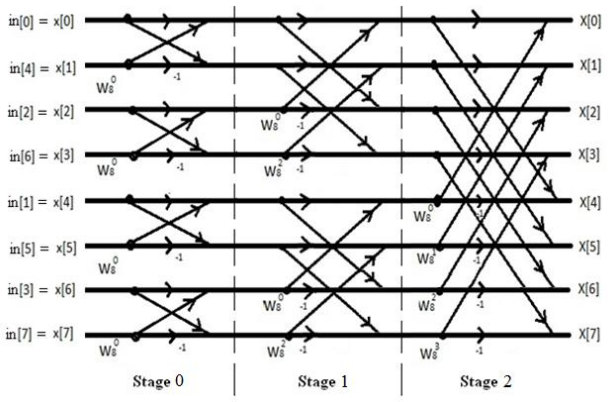

Figure 1: Radix-2 Decimation in Time (DIT)-FFT algorithm for 8-points

2.2 *CORDIC Algorithm*

CORDIC algorithm can be used to calculate the sine and cosine of an angle. To determine the sine or cosine for an angle β, the *y* or *x*-coordinate of a point on the unit circle corresponding to the desired angle must be found. Let us start with the unit vector v_0 aligned with *x*-axis:

$$v_0 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

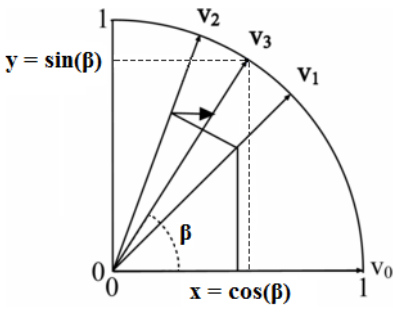

Figure 2: Illustration of CORDIC Algorithm (Wikipedia, 2014)

In the first iteration, this vector is rotated 45° counter-clockwise to get the vector v_1. Successive iterations rotate the vector in one or the other direction by size-decreasing steps, until the desired angle has been achieved. Step *i* size is $\tan^{-1}\left(\frac{1}{2^{i-1}}\right)$ for *i* = 1, 2, 3……

More formally, every step of the iteration calculates a rotation, which is performed by multiplying the vector v_{i-1} with the rotation matrix R_i:

$$v_i = R_i v_{i-1} \cdots\cdots\cdots\cdots (2)$$

The rotation matrix is given by:

$$R_i = \begin{bmatrix} \cos\gamma_i & -\sin\gamma_i \\ \sin\gamma_i & \cos\gamma_i \end{bmatrix}$$

$$= \frac{1}{\sqrt{1+(\tan\gamma_i)^2}} \begin{bmatrix} 1 & -\tan\gamma_i \\ \tan\gamma_i & 1 \end{bmatrix}$$

The equation (2) then becomes,

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} = \frac{1}{\sqrt{1+(\tan\gamma_i)^2}} \begin{bmatrix} 1 & -\tan\gamma_i \\ \tan\gamma_i & 1 \end{bmatrix} \begin{bmatrix} x_{i-1} \\ y_{i-1} \end{bmatrix}$$

where, x_{i-1} and y_{i-1} are the components of v_{i-1}. Restricting the angles γ_i so that $\tan\gamma_i$ takes on the values $\pm 2^{-i}$, the multiplication with the tangent can be replaced by a division by a power of two, which is efficiently done in digital computer hardware using a bit shift. The expression then becomes:

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} = K_i \begin{bmatrix} 1 & -\sigma_i 2^{-i} \\ \sigma_i 2^{-i} & 1 \end{bmatrix} \begin{bmatrix} x_{i-1} \\ y_{i-1} \end{bmatrix} \dots\dots\dots\dots\dots (3)$$

where,

$$K_i = \frac{1}{\sqrt{1+2^{-2i}}}$$

and σ_i can have the values of -1 or 1, and is used to determine the direction of the rotation. If the required angle β is greater than angle reached by rotation at a particular iteration, then σ_i is +1, otherwise it is -1. K_i can be ignored in the iterative process and then applied afterward with a scaling factor:

$$K(n) = \prod_{i=0}^{n-1} K_i = \prod_{i=0}^{n-1} \frac{1}{\sqrt{1+2^{-2i}}}$$

which is calculated in advance and stored in a table, or as a single constant if the number of iterations is fixed. This correction could also be made in advance, by scaling v_0 and hence saving a multiplication (Wikipedia, 2014).

3. Algorithm

As shown in Figure 1, the 2-point FFT block is the heart of FFT calculation. Each cross is the representation of one such block. Instead of using a different block to perform the cross-computation, only one cross-computation block is used and all cross-computations are carried out by that single block. The algorithm controls the inputs to the cross-computation block by checking the current stage of the FFT computation and the position of the cross within the current stage. The terms: stage of computation and position of cross within stage, are explained in detail below.

An N-point FFT computation can be divided into $\log_2(N)$ independent stages in which each stage contains $N/2$ number of cross-computations. Referring to Figure 1, the 8-point FFT is divided into $\log_2(8) = 3$ stages and each stage has $8/2 = 4$ cross-computations. The first stage of the calculation is performed on the input values. The output from each stage becomes the input to the successive stage. Also, the cross-computations within a particular stage are independent of each other. The algorithm presented here, makes use of these factors and manages to perform all computations using a single cross-computation block.

The position of cross within a stage is counted from top to bottom just like as they occur in the schematic representation of the algorithm. Referring to Figure 1, the 3-stages and crosses within each stage for 8-point FFT are shown in Table 1. The values are arranged in the form (a, b) where a *and* b are the indexes of the input pair to the cross-computation block.

Table 1: The division of stages and crosses for 8-point FFT

	Stage 0	Stage 1	Stage 2
Cross 0	(0,1)	(0,2)	(0,4)
Cross 1	(2,3)	(1,3)	(1,5)
Cross 2	(4,5)	(4,6)	(2,6)
Cross 3	(6,7)	(5,7)	(3,7)

Before Stage 0 of the computation, the re-ordering of the N input points is carried out. This is done by mirroring the $\log_2 N$ bit binary value of index of the inputs. Referring to Figure 1, the mirroring is done before stage 1 wherein the $\log_2(8) = 3$ bit index of input values $in[\dots]$ are mirrored and mapped to $x[\dots]$ as follows:

Table 2: Arranging the input values by mirroring

Decimal		Binary	
in[0]	x[0]	in[000]	x[000]
in[4]	x[1]	in[100]	x[001]
in[2]	x[2]	in[010]	x[010]
in[6]	x[3]	in[110]	x[011]
in[1]	x[4]	in[001]	x[100]
in[5]	x[5]	in[101]	x[101]
in[3]	x[6]	in[011]	x[110]
in[7]	x[7]	in[111]	x[111]

Referring to Table 1, in the Stage 0 the difference in the index of values in each crossing pair is 1. In Stage 1, in each crossing pair the difference in index of values in the pair is 2 and in Stage 3 it is 4. This means

that the value of difference in index of crossing pairs is same within a particular stage and is a function of the stage number. So, for any Stage S the value of the difference in index of crossing pairs is given by 2^S.

From Table 1, a pattern in which the crossing pairs repeat can be observed. To illustrate the repeating pattern of crosses, consider the crossing pairs in Stage 1. In Stage 1 the crossing pair Cross 0 has indexes (0, 2) and Cross 1 has indexes (1, 3). In the same stage Cross 2 has indexes (4, 6) and Cross 3 has indexes (5, 7). From these pairs, we can see that Cross 0 and Cross 1 have unique indexes and Cross 2 and Cross 3 are scaled copies of Cross 0 and Cross 1. By adding 4 to the indexes of Cross 0, Cross 2 can be obtained. Similarly, by adding 4 to the indexes of Cross 1, Cross 3 can be obtained. Also it is noted the pattern in Stage 1 repeats after every 2 crossing pairs.

This pattern can be generalized as follows:

- There are 2^S unique crossing pairs in each Stage S. All other crossing pairs in Stage S are scaled copies of these unique crossing pairs.
- The index of the first crossing pair Cross 0 for Stage S is given by $(0, 2^S)$.
- The indexes of all the unique crossing pairs can be obtained by adding p to the indexes of Cross 0 i.e. $(0 + p, 2^S + p)$ where, p ranges from 1 to $(2^S - 1)$.
- If the crossing pattern ends at Cross $(C - 1)$, then the next instance of the pattern begins at Cross C and the indexes of this instance are obtained by adding $2C$ to the indexes of the unique crossing pairs.
- For each cross-computation stage the twiddle factor is calculated by the CORDIC processor block. The angle value input to the CORDIC processor is $p\pi/2^S$ which is implemented as $p\pi \gg S$.

It is also noted that the index of the terms that are obtained at the output of each term is maintained the same as the index of the input terms. This is crucial for the working of this algorithm.

3.1 Verification of Generalizations

The values in Table 3 show how the indexes are grouped at each cross of the each of the 4 stages of 16-point FFT calculation. Following Table 3, a detailed verification of the generalizations used in the algorithm is presented.

Table 3: The division of stages and crosses for 16-point FFT

	Stage 0	Stage 1	Stage 2	Stage 3
Cross 0	(0,1)	(0,2)	(0,4)	(0,8)
Cross 1	(2,3)	(1,3)	(1,5)	(1,9)
Cross 2	(4,5)	(4,6)	(2,6)	(2,10)
Cross 3	(6,7)	(5,7)	(3,7)	(3,11)
Cross 4	(8,9)	(8,10)	(8,12)	(4,12)
Cross 5	(10,11)	(9,11)	(9,13)	(5,13)
Cross 6	(12,13)	(12,14)	(10,14)	(6,14)
Cross 7	(14,15)	(13,15)	(11,15)	(7,15)

For 16-point FFT, $N = 16$

Number of independent stages in FFT-calculation,

$$\log_2(N) = \log_2(16) = 4$$

Number of cross-computations in each stage,

$$\frac{N}{2} = \frac{16}{2} = 8$$

So, there are 4 independent stages (Stage 0 – Stage 3) with 8 cross-computations (Cross0 – Cross 7) in each stage.

For Stage 0

- Number of stage, $S = 0$
- Number of unique crossing pairs,

$$2^S = 2^0 = 1$$

So, the pattern repeats after every 1 crossing pair.

- First crossing pair in Stage 0,

$$Cross\ 0 = (0, 2^S) = (0, 1)$$

This is the only unique crossing pair in this stage. The 7 remaining crossing pairs are scaled copies of this crossing pair.

Here, crossing pattern has ended at Cross 0. The next instance of the pattern begins at Cross 1 (C = 1).

- Indexes in Cross 1 are scaled by

$$2C = 2 \times 1 = 2$$

- Second crossing pair in Stage 0 is scaled copy of Cross 0,

$$Cross\ 1 = (0 + 2C, 1 + 2C) = (0 + 2, 1 + 2) = (2, 3)$$

The next instance of the pattern begins at Cross 2 (C = 2).

- Indexes in Cross 2 are scaled by

$$2C = 2 \times 2 = 4$$

- Third crossing pair in Stage 0 is scaled copy of Cross 0,

$$Cross\ 2 = (0 + 2C, 1 + 2C) = (0 + 4, 1 + 4) = (4, 5)$$

Similarly, the indexes of crossing pairs Cross 3 to Cross 7 for Stage 0 can be calculated.

For Stage 1

- Number of stage, $S = 1$
- Number of unique crossing pairs,

$$2^S = 2^1 = 2$$

So, the pattern repeats after every 2 crossing pairs.

- First crossing pair in Stage 1,

$$Cross\ 0 = (0, 2^S) = (0, 2)$$

This is not the only unique crossing pair in this stage. The second unique crossing pair (Cross 1) can be obtained by adding p to the indexes of Cross 0.

- Range of p,

$$1\ to\ (2^S - 1) = 1\ to\ (2 - 1) = 1\ to\ 1$$

- Second crossing pair in Stage 1,

$$Cross\ 1 = (0 + p, 2^S + p) = (0 + 1, 2 + 1) = (1, 3)$$

These are the 2 unique crossing pairs in this stage. The 6 remaining crossing pairs are scaled copies of these 2 crossing pairs.

Here, crossing pattern has ended at Cross 1. The next instance of the pattern begins at Cross 2 (C = 2).

- Indexes in Cross 2 and Cross 3 are scaled by

$$2C = 2 \times 2 = 4$$

- Third crossing pair in Stage 1 is scaled copy of Cross 0,

$$Cross\ 2 = (0 + 2C, 2 + 2C) = (0 + 4, 2 + 4) = (4, 6)$$

- Fourth crossing pair in Stage 1 is scaled copy of Cross 1,

$$Cross\ 3 = (1 + 2C, 3 + 2C) = (1 + 4, 3 + 4) = (5, 7)$$

Here, crossing pattern has ended at Cross 3. The next instance of the pattern begins at Cross 4 (C = 4).

- Indexes in Cross 4 and Cross 5 are scaled by

$$2C = 2 \times 4 = 8$$

- Fifth crossing pair in Stage 1 is scaled copy of Cross 0,

$$Cross\ 4 = (0 + 2C, 2 + 2C) = (0 + 8, 2 + 8) = (8, 10)$$

- Sixth crossing pair in Stage 1 is scaled copy of Cross 1,

$$Cross\ 5 = (1 + 2C, 3 + 2C) = (1 + 8, 3 + 8) = (9, 11)$$

Similarly, the indexes of crossing pairs Cross 6 and Cross 7 for Stage 1 can be calculated.

For Stage 2

- Number of stage, $S = 2$
- Number of unique crossing pairs,

$$2^S = 2^2 = 4$$

So, the pattern repeats after every 4 crossing pairs.

- First crossing pair in Stage 2,

$$Cross\ 0 = (0, 2^S) = (0, 4)$$

This is not the only unique crossing pair in this stage. The 3 other unique crossing pair (Cross 1, Cross 2 and Cross 3) can be obtained by adding p to the indexes of Cross 0.

- Range of p,

$$1\ to\ (2^S - 1) = 1\ to\ (4 - 1) = 1\ to\ 3$$

- Second crossing pair in Stage 2,

$$Cross\ 1 = (0 + p, 2^S + p) = (0 + 1, 4 + 1) = (1, 5)$$

- Third crossing pair in Stage 2,

$$Cross\ 2 = (0 + p, 2^S + p) = (0 + 2, 4 + 2) = (2, 6)$$

- Fourth crossing pair in Stage 2,

$$Cross\ 3 = (0 + p, 2^S + p) = (0 + 3, 4 + 3) = (3, 7)$$

These are the 4 unique crossing pairs in this stage. The 4 remaining crossing pairs are scaled copies of these two crossing pairs.

Here, crossing pattern has ended at Cross 3. The next instance of the pattern begins at Cross 4 (C = 4).

- Indexes in Cross 4 to Cross 7 are scaled by

$$2C = 2 \times 4 = 8$$

- Fifth crossing pair in Stage 2 is scaled copy of Cross 0,

$$Cross\ 4 = (0 + 2C, 4 + 2C) = (0 + 8, 4 + 8) = (8, 12)$$

- Sixth crossing pair in Stage 2 is scaled copy of Cross 1,

$$Cross\ 5 = (1 + 2C, 5 + 2C) = (1 + 8, 5 + 8) = (9, 13)$$

- Seventh crossing pair in Stage 2 is scaled copy of Cross 2,

$$Cross\ 6 = (2 + 2C, 6 + 2C) = (2 + 8, 6 + 8) = (10, 14)$$

- Eighth crossing pair in Stage 2 is scaled copy of Cross 3,

$$Cross\ 7 = (3 + 2C, 7 + 2C) = (3 + 8, 7 + 8) = (11, 15)$$

For Stage 3

- Number of stage, $S = 3$
- Number of unique crossing pairs,

$$2^S = 2^3 = 8$$

So, the pattern repeats after every 4 crossing pairs.

- First crossing pair in Stage 3,

$$Cross\ 0 = (0, 2^S) = (0, 8)$$

This is not the only unique crossing pair in this stage. The 8 other unique crossing pair (Cross 1 to Cross 7) can be obtained by adding p to the indexes of Cross 0.

- Range of p,

$$1\ to\ (2^S - 1) = 1\ to\ (8 - 1) = 1\ to\ 7$$

- Second crossing pair in Stage 3,

$$Cross\ 1 = (0 + p, 2^S + p) = (0 + 1, 8 + 1) = (1, 9)$$

- Third crossing pair in Stage 3,

$$Cross\ 2 = (0 + p, 2^S + p) = (0 + 2, 8 + 2) = (2, 10)$$

Similarly, the indexes of crossing pairs Cross 3 and Cross 7 for Stage 3 can be calculated.

4. Experiment for Testing

For testing the algorithm presented in this paper, an HDL module was developed using Verilog HDL. The module was coded using Xilinx ISE v14.7 and verified using the ISim Simulator. The designed module takes the value of N (for N-point FFT) as a parameter input during module instantiation. So, the designed module can be scaled to any N-point FFT-computation core by simply changing the value of parameter N at module instantiation. The default value for the parameter N is set to 16 for 16-point FFT and this default is used to generate the results presented in section 5.

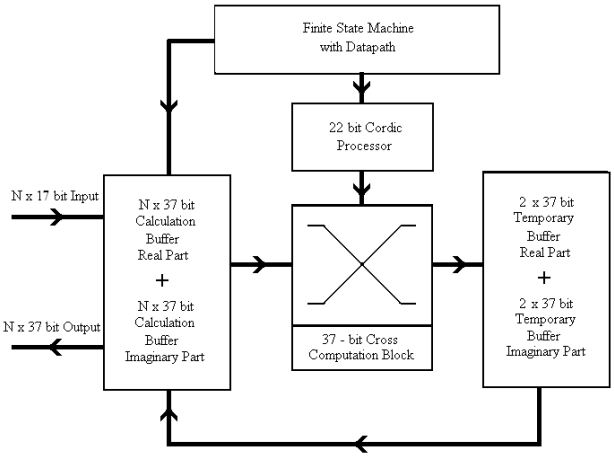

Figure 3: Block Diagram of Designed System

The designed module takes N number of 17 bit signed values as input. As all the calculations are performed in fixed point, so the inputs are scaled by a factor of 2^{20}(implemented as left shift by 20) and are arranged into two Nx37 bit register file buffers, one storing real values and the other storing imaginary values. The Finite State Machine with Data-path (FSMD) passes the indexes of crossing pairs, one pair at a time, to the register buffers to select values to pass to the cross computation block. The cross computation block performs the Radix-2 calculation for the values at its input and stores the results in a temporary buffer. The values from the temporary buffer are then written back to the calculation buffer at the same indexes as those of the crossing pair input to the cross-computation block.

The twiddle factors required to perform the FFT cross-computation are supplied to the cross-computation block by the CORDIC processor unit. The value of angle, required for twiddle factor calculation at each cross-computation, is determined within the FSMD and passed to the CORDIC processor. The CORDIC processor uses the angle input to it to generate the twiddle factors.

Next, brief descriptions of all of the functional blocks of the designed module are presented.

4.1 Cross – Computation Block

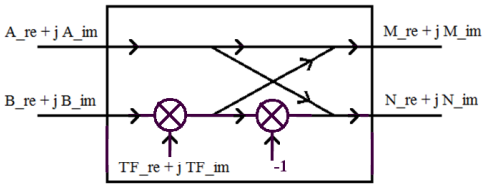

Figure 4: Cross Computation Block

The cross-computation block used in the design takes two input terms each with a real and imaginary part. The data-width of the input terms is 37-bit. Referring to Figure 4 the input terms are A_{re}, A_{im}, B_{re} and B_{im}. The twiddle factor, with a data width of 22-bit, is supplied by the CORDIC processor and is also an input to the cross-computation block. In Figure 4 the twiddle factor terms are T_{re} and T_{im}. There are two outputs terms from the cross-computation block, each 37-bit wide which are represented as M_{re}, M_{im}, N_{re} and N_{im} in Figure 4.

The cross-computation block is designed using combinational logic, so the calculations performed within this block do not use up any clock cycles. The calculations performed by the cross computation block are shown below:

$$M_{re} = A_{re} + (B_{re}.T_{re} - B_{im}.T_{im})$$
$$M_{im} = A_{re} + (B_{re}.T_{im} - B_{im}.T_{re})$$
$$N_{re} = A_{re} - (B_{re}.T_{re} - B_{Im}.T_{Im})$$
$$N_{im} = A_{re} - (B_{re}.T_{im} - B_{im}.T_{re})$$

Calculating the terms $(B_{re}.T_{re} - B_{im}.T_{im})$ and $(B_{re}.T_{im} - B_{im}.T_{re})$ requires the use of 4 multiplier units and 2 adder/subtractor units. Since a multiplier unit occupies larger design area, it is best practice to reduce the number of multipliers in digital design. To this end, in this paper an alternative method to perform the above calculations is adopted. This method reduces the number of multiplier units to 3.

The alternative method is presented below:

$$intrm = B_{re}.(T_{re} + T_{im})$$
$$I_{re} = intrm - T_{im}(B_{re} + B_{im})$$
$$= (B_{re}.T_{re} - B_{im}.T_{im})$$
$$I_{im} = intrm - T_{re}.(B_{re} - B_{im})$$
$$= (B_{re}.T_{im} - B_{im}.T_{re})$$
$$M_{re} = A_{re} + I_{re}$$
$$M_{im} = A_{im} + I_{im}$$
$$N_{re} = A_{re} - I_{re}$$
$$N_{im} = A_{im} - I_{im}$$

It is noted that only one instance of the cross-computation block is defined in the code. All the cross-computation operations are performed by that one instance.

4.2 CORDIC Processor

The CORDIC processor block used in the design takes a 22-bit radian angle measure as input and returns the value of sine and cosine scaled by a factor of 2^{20} in two 22-bit output ports. Referring to Figure 5 the input term is $Angle\_rad$ and the output terms are Val_cos and Val_sin. The outputs from the CORDIC processor are directly interfaced to the twiddle factor inputs of the cross-computation block.

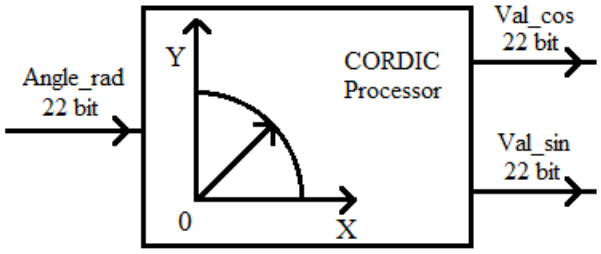

Figure 5: CORDIC Processor Block

The CORDIC processor is designed with memory of 7 values of $\tan\gamma_i$ (Refer Section 2). The value of sine and cosine are output from a chain of 20-iterations. When separately tested, the CORDIC processor block exhibited a precision of 5 points after decimal.

The CORDIC processor block is also modeled using combinational logic, which means that it also does not use up any clock cycles for calculation.

4.3 Finite State Machine with Data-path (FSMD)

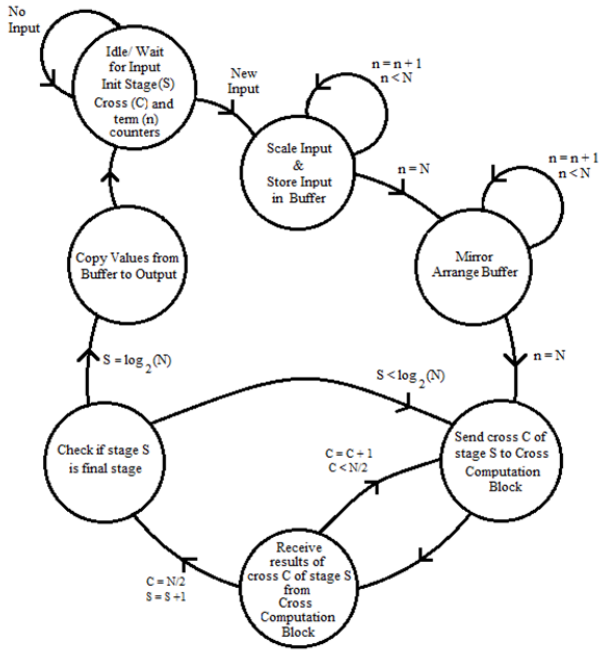

Figure 6: Schematic representation of the states of the FSMD

The Finite State Machine with Data-path (FSMD) used in the design manages all the operations performed by

the designed module. The FSMD performs tasks such as storing input values to buffer, routing values from buffer to cross-computation block and managing the counters for current stage of FFT and current cross within that stage. The FSMD is designed with 7 states. The steps involved in transition from one state to another are explained below:

State 1 of FSMD (Idle / Wait for Input)

The first state of the FSMD is the idle state. In this state the FSMD waits for new inputs to arrive. In this state the values for counters of State (S), Cross (C) and term (n) are initialized. When the signal for new input is received then the FSMD changes to state 2.

State 2 of FSMD (Scale Input and Store in Buffer)

In the second state the inputs are read from the input ports of the module and are stored in the Nx37-bit calculation buffer after scaling by the factor 2^{20}(Refer to Figure 3). The indexing for the buffer is done using the term n. The FSMD cycles back to state 2 till the term n is equal to the number of inputs N, to complete the data scale and store operation.

State 3 of FSMD (Mirror Arrange Buffer)

In this stage, the scaled inputs in the buffer storage are re-ordered according to the principle previously explained in Section 3 and Table 2 of this paper. The mirroring operation also uses the term n for indexing the values stored in the buffer. So the FSMD cycles back to state 3 till the term n is equal to the number of values N. Note that a separate temporary buffer is used to store the mirrored values while the FSMD loops within state 3.

State 4 of FSMD (Send Cross C of Stage S to Cross-Computation Block)

In this state, a number of counters for stage S, cross C, range p (Refer to Section 3 Verification of Generalizations) are used to calculate the index of values to send to the cross-computation block. The value of $Angle\_rad$ (Refer Section 4.2) is also determined in this state and passed to the CORDIC processor module.

State 5 of FSMD (Receive results of Cross C of Stage S to Cross-Computation Block)

In this state, the results from the temporary buffer (Refer Figure 3) are written back to the calculation buffer at the same indexes as the input to cross-computation block. Then, the value in counter C is checked to determine whether it is final cross for Stage S. If the counter indicates that it is not the final cross then the cross counter is incremented and the FSMD loops to state 4. If the counter indicates that it is the final cross then the stage counter S is incremented by 1 and the FSMD transitions to state 6.

State 6 of FSMD (Check if Stage S is final stage)

The value of counter S is checked to determine whether or not Stage S is the final stage of FFT-computation. If the counter indicates that it is not the final stage of the FFT-computation then the FSMD loops to stage 4. If the counter indicates that it is the final stage of the FFT-computation then the FSMD transitions to state 7.

State 7 of FSMD (Copy Values from Buffer to Output)

When state 7 of the FSMD is reached then the results of the FFT-computation are copied from the calculation buffer to the output ports of the designed module. After doing so, the FSMD transitions back to state 1 and waits for new input to arrive.

5. Result

Table 4 shows the result of applying our algorithm to calculate 16-point FFT. The result obtained by applying proposed algorithm has been compared with the actual FFT values obtained from MATLAB. The table clearly shows that our algorithm calculates FFT with high precision. The absolute value of error is always below 0.03 for the input values of 3 digits. Therefore, proposed algorithm is precise up to 4 digits.

The simulation waveform in Figure 7 shows the time for calculation of 16-point FFT. As shown by the figure, the time for calculation is only 2µs. This is a crucial result, obtained from the experiment, because time required for calculation is well within the tolerable range, despite the fact that we have performed all the computations using a single cross-computation block. Thus, reusing same hardware for all computation minimizes the hardware resources, uses minimum chip area as well as considerably simplifies the hardware design. Yet, the computation time is well within the acceptable range.

Table 4: Result showing precision of FFT calculation

Index	Input Signal	FFT from MATLAB (a)	FFT from our algorithm (b)	Difference d = a - b	Absolute Error |d|
0	100	7200	7199.986267	0.0137333	0.013732
1	200	-2527.414237 -i*502.733949	-2527.386168 -i*502.737194	-0.0280684 +i*0.003245	0.028255
2	300	0	0.000476 -i*0.001149	-0.000476 +i*0.001149	0.0012438
3	400	-223.982881 -i*149.660576	-223.989063 -i*149.652841	0.006182 -i*0.007735	0.0099
4	500	0	0.000475 -i*0.000197	-0.000475 +i*0.000197	0.0018868
5	600	-44.646269 -i*66.817864	-44.651729 -i*66.80841	0.005459 -i*0.009454	0.010917
6	700	0	0.000475 -i*0.000197	-0.000475 +i*0.000197	0.00051
7	800	-3.956613 -i*19.891237	-3.970139 -i*19.889966	0.013526 -i*0.001271	0.013585
8	800	0	0.003051	-0.003051	0.003051
9	700	-3.956613 -i*19.891237	-3.969812 +i*19.891671	0.013199 -i*0.000434	0.0132066
10	600	0	0.000475 +i*0.000196	-0.000475 -i*0.000196	0.0005139
11	500	-44.646269 -i*66.817864	-44.65191 +i*66.808463	0.005641 +i*0.0094	0.0109631
12	400	0	0.001334 +i*0.001333	-0.001334 -i*0.001333	0.001886
13	300	-223.982881 -i*149.660576	-223.989243 +i*149.652784	0.006362 +i*0.007792	0.010059
14	200	0	0.000476 +i*0.0011463	-0.000476 -i*0.0011463	0.0012412
15	100	-2527.414237 -i*502.733949	-2527.38584 +i*502.735481	-0.028396 +i*0.001532	0.0284378

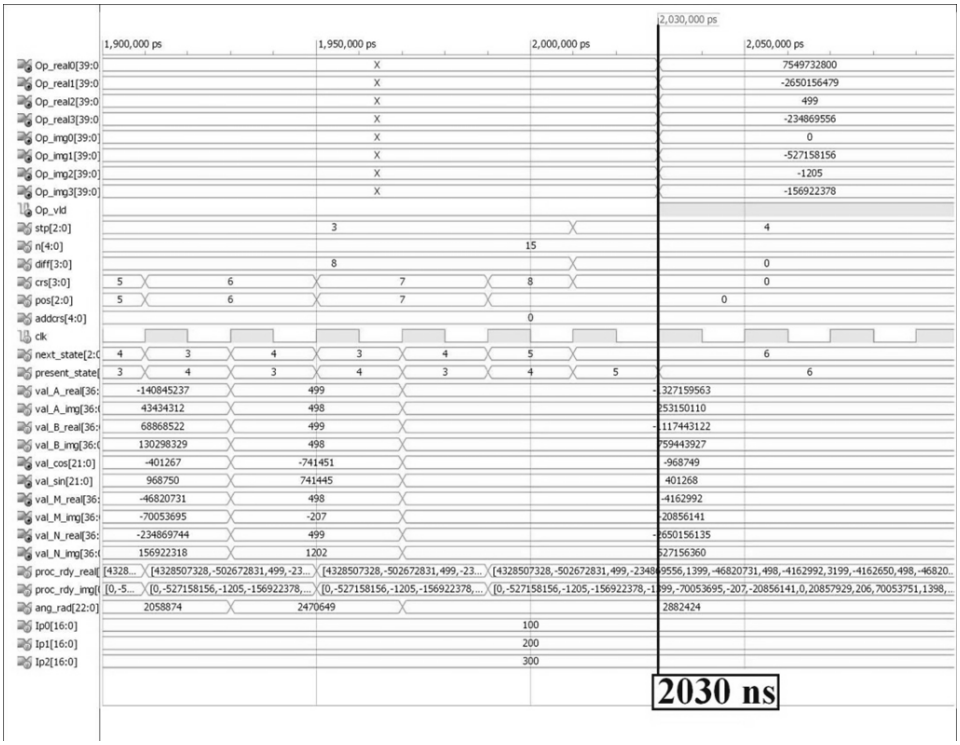

Figure 7: Simulation Waveform from test of 16-point FFT

Consider the calculation of FFT for audio signals. Audio signals range in the frequency of 0 to 44 kHz and the Nyquist rate for sampling is below 100 kHz. Even if the audio signal is sampled at twice the Nyquist rate, sampling frequency becomes 200 kHz and sampling period will be 5 microsecond. Thus, calculation time of 2µs is well within the required range for audio processing.

Therefore, the results corroborate our claim that our design and algorithm keeps the architecture simple and minimizes resources without increasing the computation time beyond acceptable range.

6. Conclusion

The algorithm presented in this paper calculates FFT with high precision, with satisfactory speed and at the same time minimizes hardware resources. In addition, our algorithm is highly generalized and can be used to calculate any N-point FFT by changing only the value of N during module instantiation.

References

Oppenheim, A. V., Schafer, R. W., & Buck, J. R. (2012). *Discrete-Time Signal Processing* (Second ed.). Pearson Education.

Wikipedia. (2014, August). *Cooley–Tukey FFT algorithm*. Retrieved August 2014, from Wikipedia: http://en.wikipedia.org/wiki/Cooley%E2%80%93Tukey_FFT_algorithm

Wikipedia. (2014, Septembeer). *CORDIC*. Retrieved August 2014, from Wikipedia: http://en.wikipedia.org/wiki/CORDIC