

Implementation of Fast Fourier Transform (FFT) on FPGA using Verilog HDL

An Advanced-VLSI-Design-Lab (AVDL) Term-Project,
VLSI Engineering Course, Autumn 2004-05,
Deptt. Of Electronics & Electrical Communication,
Indian Institute of Technology Kharagpur

Under the guidance of
Prof. Swapna Banerjee
Deptt. Of Electronics & Electrical Communication Engg.
Indian Institute of Technology Kharagpur.

Submitted by
Abhishek Kesh (02EC1014)
Chintan S.Thakkar (02EC3010)
Rachit Gupta (02EC3012)
Siddharth S. Seth (02EC1032)
T. Anish (02EC3014)

ACKNOWLEDGEMENTS

It is with great reverence that we wish to express our deep gratitude towards our VLSI Engineering Professor and Faculty Advisor, **Prof. Swapna Banerjee**, Department of Electronics & Electrical Communication, Indian Institute of Technology Kharagpur, under whose supervision we completed our work. Her astute guidance, invaluable suggestions, enlightening comments and constructive criticism always kept our spirits up during our work.

We would be accused of ingratitude if we failed to mention the consistent encouragement and help extended by Mr. Kailash Chandra Ray, Graduate Research Assistant, during our Term-Project work. The brainstorming sessions at AVDL spent discussing various possible architectures for the FFT were very educative for us novice VLSI students.

Our experience in working together has been wonderful. We hope that the knowledge, practical and theoretical, that we have gained through this term project will help us in our future endeavours in the field of VLSI.

Abhishek Kesh
Chintan S.Thakkar
Rachit Gupta
Siddharth S. Seth
T. Anish

1. FAST FOURIER TRANSFORMS

The number of complex multiplication and addition operations required by the simple forms both the Discrete Fourier Transform (DFT) and Inverse Discrete Fourier Transform (IDFT) is of order N^2 as there are N data points to calculate, each of which requires N complex arithmetic operations.

For length n input vector x , the DFT is a length n vector X , with n elements:

$$f_j = \sum_{k=0}^{n-1} x_k e^{-(2\pi i/n)jk} \quad j = 0, \dots, n-1.$$

In computer science jargon, we may say they have algorithmic complexity $O(N^2)$ and hence is not a very efficient method. If we can't do any better than this then the DFT will not be very useful for the majority of practical DSP applications. However, there are a number of different 'Fast Fourier Transform' (FFT) algorithms that enable the calculation the Fourier transform of a signal much faster than a DFT.

As the name suggests, FFTs are algorithms for quick calculation of discrete Fourier transform of a data vector. The FFT is a DFT algorithm which reduces the number of computations needed for N points from $O(N^2)$ to $O(N \log N)$ where \log is the base-2 logarithm. If the function to be transformed is not harmonically related to the sampling frequency, the response of an FFT looks like a 'sinc' function $(\sin x) / x$

The 'Radix 2' algorithms are useful if N is a regular power of 2 ($N=2^p$). If we assume that algorithmic complexity provides a direct measure of execution time and that the relevant logarithm base is 2 then as shown in Fig. 1.1, ratio of execution times for the (DFT) vs. (Radix 2 FFT) (denoted as 'Speed Improvement Factor') increases tremendously with increase in N .

The term 'FFT' is actually slightly ambiguous, because there are several commonly used 'FFT' algorithms. There are two different Radix 2 algorithms, the so-called 'Decimation in Time' (DIT) and 'Decimation in Frequency' (DIF) algorithms. Both of these rely on the recursive decomposition of an N point transform into 2 ($N/2$) point transforms. This decomposition process can be applied to *any composite* (non prime) N . The method is particularly simple if N is divisible by 2 and if N is a regular power of 2, the decomposition can be applied repeatedly until the trivial '1 point' transform is reached.

Number of Points, N	Complex Multiplications in Direct Computation, N^2	Complex Multiplications in FFT Algorithm, $(N/2) \log_2 N$	Speed Improvement Factor
4	16	4	4.0
8	64	12	5.3
16	256	32	8.0
32	1,024	80	12.8
64	4,096	192	21.3
128	16,384	448	36.6
256	65,536	1,024	64.0
512	262,144	2,304	113.8
1,024	1,048,576	5,120	204.8

Fig. 1.1: Comparison of Execution Times, DFT & Radix – 2 FFT

The radix-2 decimation-in-frequency FFT is an important algorithm obtained by the divide-and-conquer approach. The Fig. 1.2 below shows the first stage of the 8-point DIF algorithm.

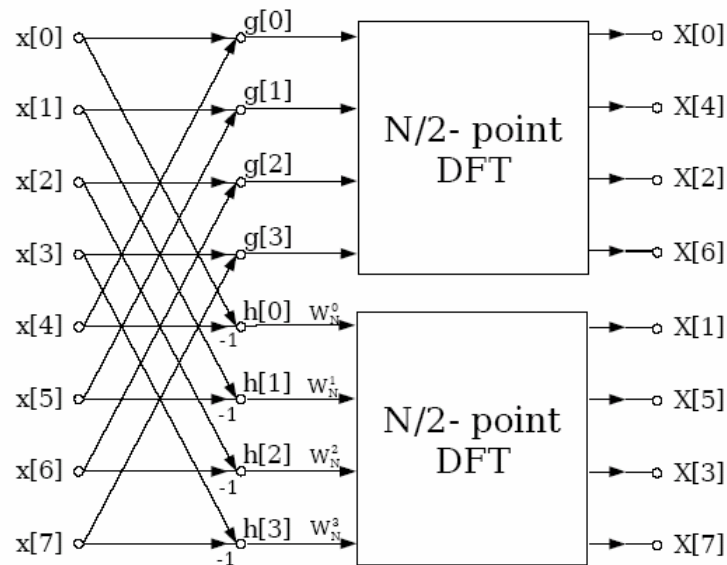


Fig. 1.2: First Stage of 8 point Decimation in Frequency Algorithm.

The decimation, however, causes shuffling in data. The entire process involves $\nu = \log_2 N$ stages of decimation, where each stage involves $N/2$ butterflies of the type shown in the Fig. 1.3.

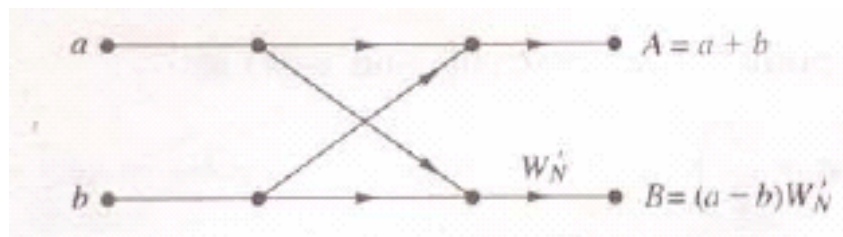


Fig. 1.3: Butterfly Scheme.

Here $W_N = e^{-j2\pi/N}$ is the Twiddle factor.

Consequently, the computation of N-point DFT via this algorithm requires $(N/2) \log_2 N$ complex multiplications. For illustrative purposes, the eight-point decimation-in frequency algorithm is shown in the Figure below. We observe, as previously stated, that the output sequence occurs in bit-reversed order with respect to the input. Furthermore, if we abandon the requirement that the computations occur in place, it is also possible to have both the input and output in normal order.

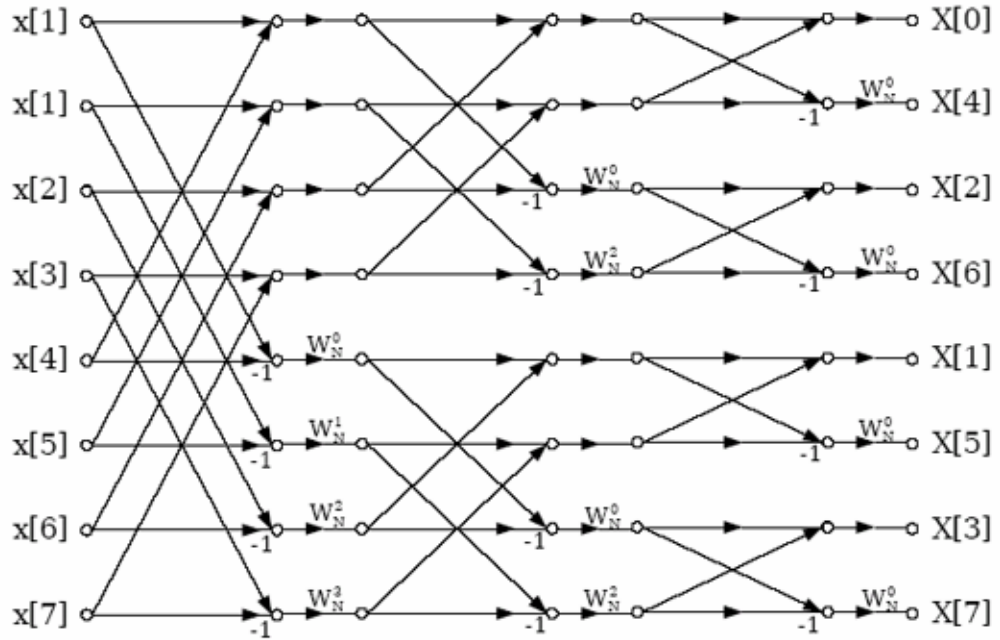


Fig. 1.4: 8 point Decimation in Frequency Algorithm

2. ARCHITECTURE

2.1 Comparative Study

Our Verilog HDL code implements an 8 point decimation-in-frequency algorithm using the butterfly structure. The number of stages v in the structure shall be $v = \log_2 N$. In our case, $N = 8$ and hence, the number of stages is equal to 3. There are various ways to implement these three stages. Some of them are,

A) Iterative Architecture - Using only one stage iteratively three times, once for every decimation

This is a hardware efficient circuit as there is only one set of 12-bit adders and subtractors. The first stage requires only 2 CORDICs. The computation of each CORDIC takes 8 clock pulses. The second and third stages do not require any CORDIC, although in this structure they will require to rotate data by 0° or -90° using the CORDIC, which will take 16 (8 for the second and 8 for third stage) clock pulses. The entire process of rotation by 0° or -90° can rather be easily achieved by 2's complement and BUS exchange which would require much less hardware. Besides, while one set of data is being computed, we have no option but to wait for it to get completely processed for 36 clock cycles before inputting the next set of data.

Thus,

Time Taken for computation = 24 clock cycles

No. of 12 bit adders and subtractors = 16

a) Pipeline Architecture - Using three separate stages, one each for every decimation

This is the other extreme which would require 3 sets of sixteen, 12-bit adders. The complexity of implementation would definitely be reduced and delay would drastically cut down as each stage would be separated from the other by a bank of registers, and one set of data could be serially streamed into the input registers 8 clock pulses after the previous set. The net effect is that at a time we can have 3 stages working simultaneously.

However, this architecture is not taken into consideration as a valid option simply because of the immense hardware required. Besides, it would give improvement of merely 1 clock cycle over the architecture discussed below which we have used in terms of the total time taken.

Thus,

Time Taken for computation = 8 clock cycles

No. of 12 bit adders and subtractions = 40

b) Proposed Method - Using 2 stages to calculate the 3 decimations

Our architecture attempts to strike a balance between the iterative and pipeline architectures. We use two stages for the 3 decimations. The first stage is implemented in standard fashion. It is the second and third stages which are merged together to form one stage, as they do not require any CORDIC. The selection of data for computation is controlled by MUX which is in turn controlled by the COUNTER MUX. The first stage requires adders and subtractors only for the REAL data, while next stage requires adders and subtractors for both REAL and IMAGINARY data.

Thus,

$$\text{Time Taken for computation} = 10 \text{ clock cycles}$$

$$\text{No. of 12 bit adders and subtractors} = 24$$

The above data clearly highlights the fact that the implemented architecture is a trade-off between the two extreme architectures.

2.2 Working

The data is serially entered into the circuit. Depending upon the output of the counter, the data goes into the respective 12 bit register for parallel input. The first 8 clock pulses are used in this input process as shown in the Fig. 2.2.1. This data later automatically acts as input to the asynchronous adders and subtractors.

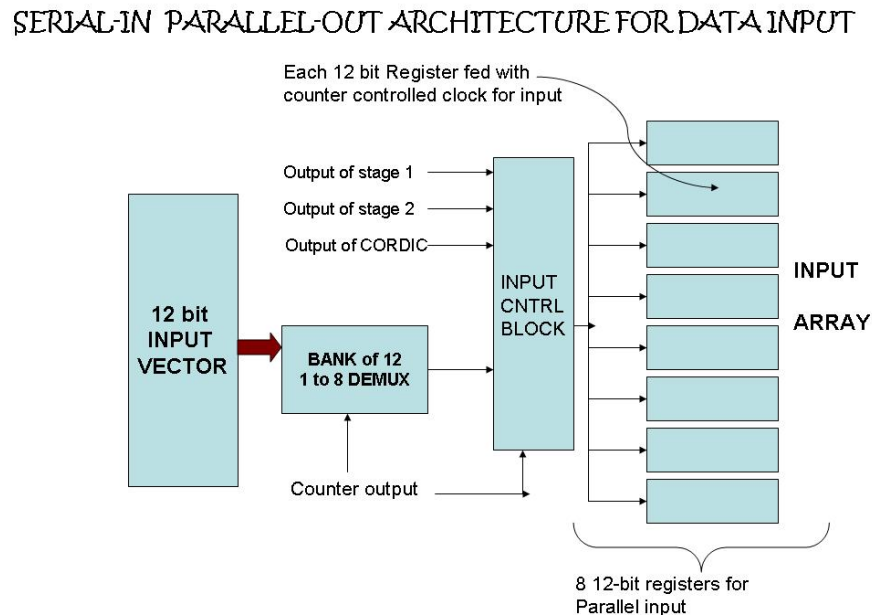


Fig. 2.2.1: Input Architecture

The outputs are now ready to be inputted to the CORDIC block. Outputs 0 to 5 and 8 are ready for next stage, but the outputs to the CORDIC are available only after 8 more clock pulses. Hence, the output to the second stage is available only after $8+8=16$ clock pulses. This output is loaded into the input register, whose output is in turn fed to stage 2 of the circuit.

The stage 2 in this circuit jointly implements both the second and third decimations in the architecture simply because there is no CORDIC required in these stages and rotation required is -90° or 0° .

Thus, $a+bj$ on rotation by -90° becomes $b-aj$, i.e. simply 2's complement of 'a'

BLOCK DIAGRAM OF BUTTERFLY STRUCTURE

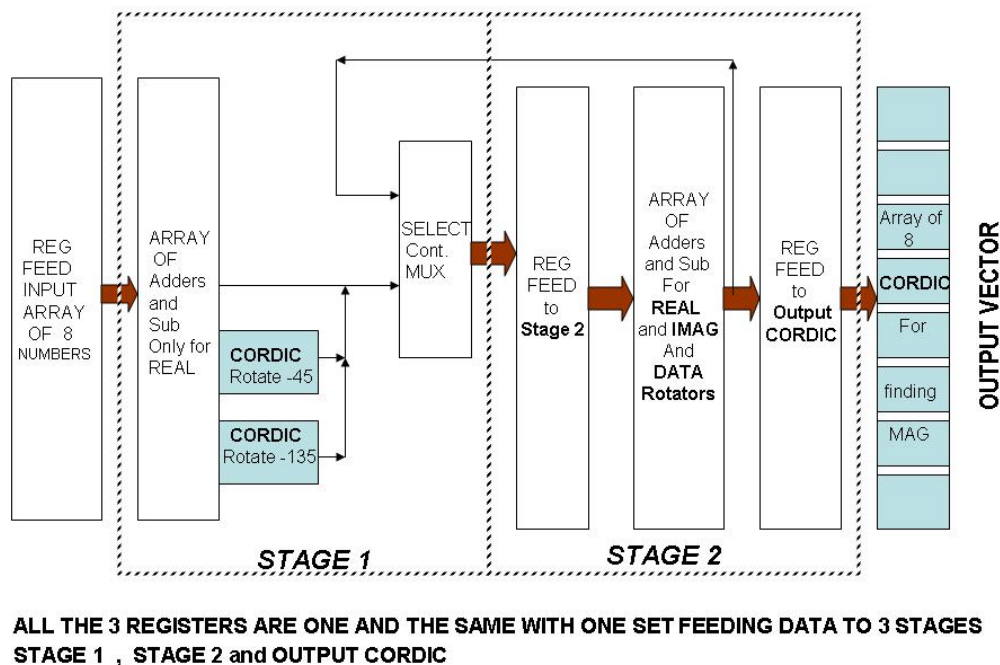


Fig. 2.2.2: Butterfly Scheme.

The Fig. 2.2.3 displays how by varying the input of data, both the stages can be implemented using only one stage and used iteratively. If the second and third inputs are flipped, we get the structure for the third stage. As both second and third stages are asynchronous, they require only one clock pulse each for computation.

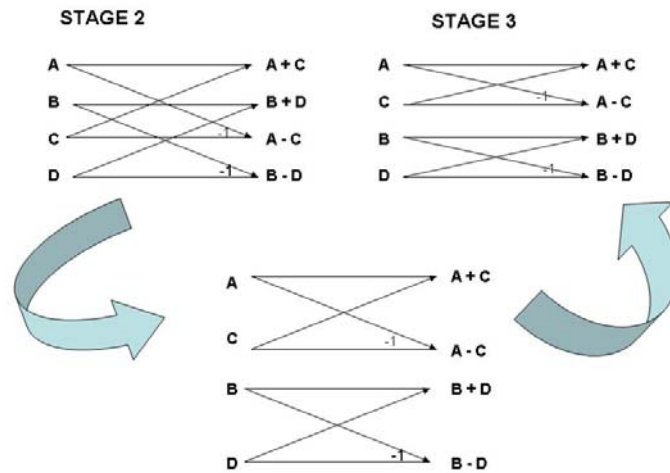


Fig. 2.2.3: Adjustments done to implement 2nd & 3rd Stage together

After we get the output at the end of the 3rd stage, it is loaded into the VECTORING CORDIC. The VECTORING CORDIC gives the magnitude of the complex number entered as $\text{Real} + \text{Imag} * j$ as the output, taking 8 clock cycles to compute.

PARALLEL-IN SERIAL-OUT ARCHITECTURE FOR DATA OUTPUT

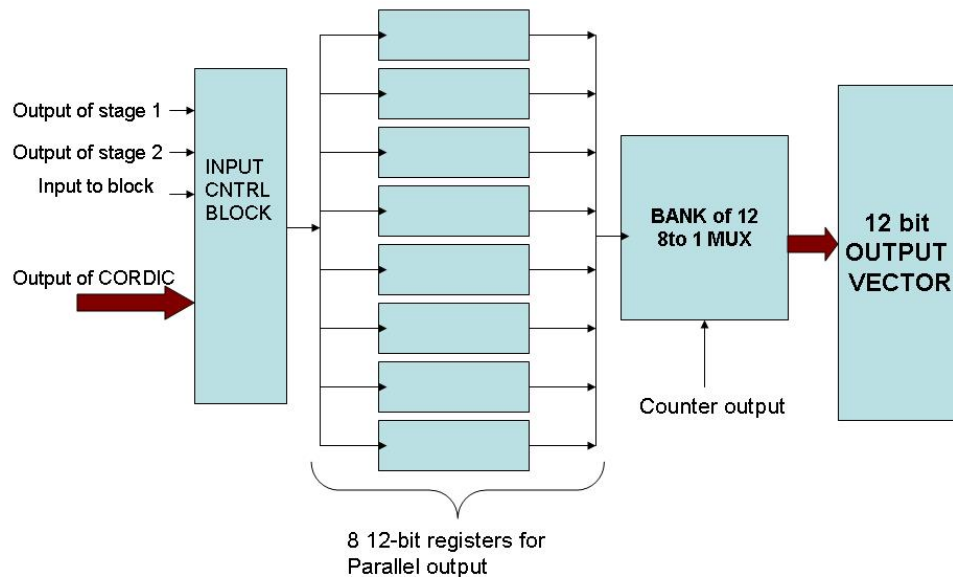


Fig. 2.2.4: Output Architecture

We then send these 8 outputs serially in the output port in the next 8 clock cycles. The above architecture illustrates how the output is channeled into a 12 bit port by the use of counter value and the bank of multiplexers.

Thus, the entire operation of taking in the input vector, performing FFT and giving the result in the output port takes a total of 34 clock cycles. The distribution is summarized as follows.

8 cycles	Taking the 8 real values into reg_x[0:7]
8 cycles	Performing Rotation CORDIC
2 cycles	2nd and 3rd Stage of Butterfly Scheme
8 cycles	Performing the Vectoring CORDIC to get the magnitude
8 cycles	Giving the 8 magnitude values into 'out' one after the other

3. Building blocks

As we saw in the last section, the FFT architecture uses certain blocks as Rotation CORDIC, Vectoring CORDIC, Twelve Bit Adder and Counters. The CORDIC blocks themselves require Shifters and registers. These blocks are now explained.

3.1 Rotation CORDIC

CORDIC is an acronym for Co-Ordinate Rotation DIgital Computer, as a basic processing element. The rotational mode of CORDIC is used only in the first stage of the butterfly scheme where we wish to rotate the input vector which is real, i.e. only x component. As such we pass only a single real value x. The output is a complex vector with both real and imaginary components.

There are two instantiations of this module. One is to compute the rotation by -45 degrees and the other by -135 degrees. Now, when the original vector is on the X axis, then we can rotate it by -45 degrees and then negate the x component to get the vector we would have got had we rotated by -135 degrees. Taking advantage of this fact, we do not pass on -45 degrees or -135 degrees to this module. The module always performs a -45 degrees rotation on the input real valued vector. The calling module then performs the required negation of the y component to get the rotation by -135 degrees.

In Rotation CORDIC, pseudo rotation takes place as shown in Fig. 3.1.1.

Rotations and Pseudo-Rotations

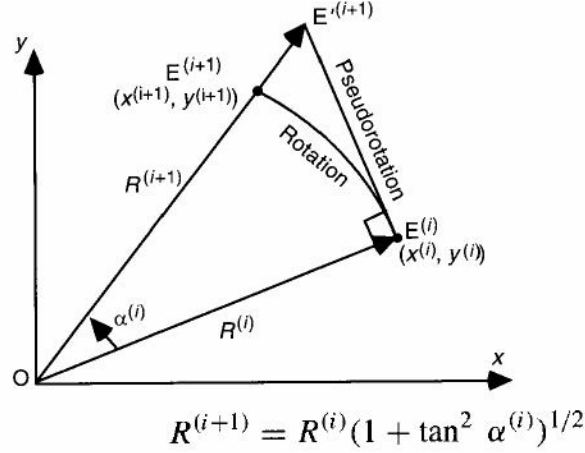


Fig. 3.1.1: Pseudo rotation

Because of this, the x and y components get multiplied by the CORDIC gain factor of 1.647. The exact gain depends on the number of iterations and obeys the relation:

$$A_n = \prod_n \sqrt{1 + 2^{-2i}}$$

To remove this factor we need to do compensation.

We have two instantiations of the rotate_cordic module running side by side. One cordic rotates the input x by angle_a = (-45 + beta) degrees and the other rotates the input x by angle_b = (-45 - beta) degrees.

Now, the outputs of these two cordics at the end of 8 iterations are:

Cordic 1: $x_a = A_n * x * \cos(\text{angle_a})$, $y_a = A_n * x * \sin(\text{angle_a})$

Cordic 2: $x_b = A_n * x * \cos(\text{angle_b})$, $y_b = A_n * x * \sin(\text{angle_b})$

where $A_n = 1.647 = \text{CORDIC gain factor (after 8 iterations.)}$

At the end of this, we get the final x and y values by taking the mean of x_a & x_b and y_a & y_b .

This compensates the CORDIC gain factor as follows:

By taking $\cos(\text{beta}) = (1/A_n)$,

$$x1 = (x_a + x_b)/2 = (A_n * x * \cos(\text{angle_a}) + A_n * x * \cos(\text{angle_b}))/2$$

$$= A_n * x * (\cos(-45 + \text{beta}) + \cos(-45 - \text{beta}))/2 = A_n * x * \cos(-45) * \cos(\text{beta}) = x * \cos(-45)$$

$$y1 = (y_a + y_b)/2 = (A_n * x * \sin(\text{angle_a}) + A_n * x * \sin(\text{angle_b}))/2$$

$$= A_n * x * (\sin(-45 + \text{beta}) + \sin(-45 - \text{beta}))/2 = A_n * x * \sin(-45) * \cos(\text{beta}) = x * \sin(-45)$$

3.1.2 Basic Rotation CORDIC Block

Uses the standard CORDIC algorithm to rotate a vector $x + jy$ by inp_angle degrees. Takes 8 iterations.

When started, the x , y and the angle registers are initiated to the original values of x , y and inp_angle respectively. Then, depending on the sign of the angle accumulated in the angle register (this sign is the sign bit $\text{angleadderin}[11]$), the x and y registers either add or subtract the shifted values of y and x respectively at every iteration. The block diagram is as follows:

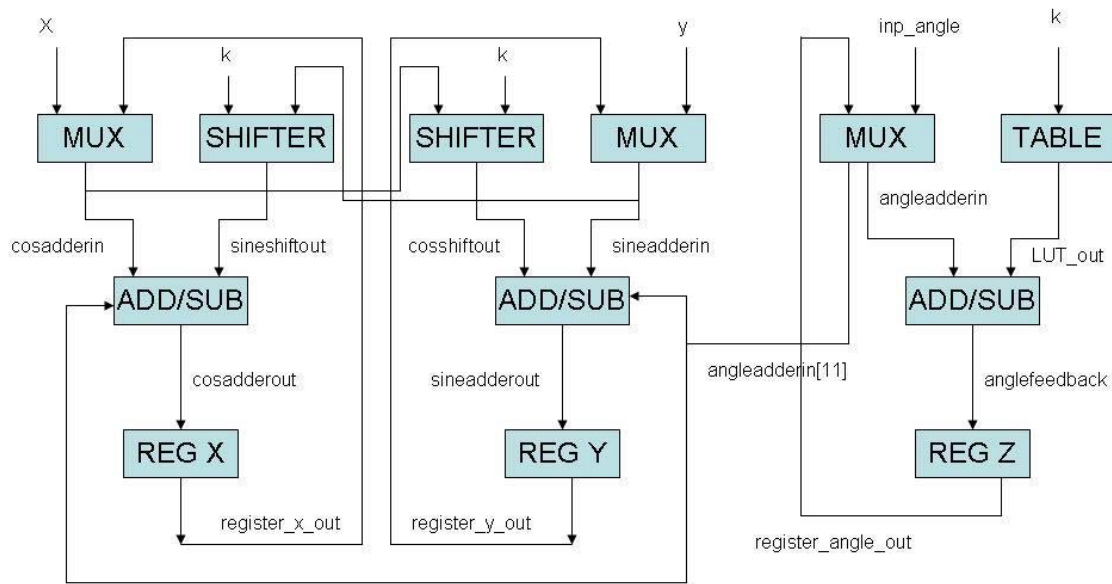


Fig. 3.1.2.1: Block Diagram, Rotation CORDIC

3.2 Vectoring CORDIC

This module makes a vectoring cordic that computes the magnitude (mag) of a vector $x + jy$.

Please note that the CORDIC gain factor compensation is not done in this vectoring CORDIC.

As such, the magnitude values are actually multiplied by the CORDIC gain Factor of 1.647.

The module uses the same concept as that explained in the rotation cordic. When started, the x and y registers are initiated with the original values of x & y respectively. Then, we try to iteratively rotate the vector so that it comes onto the x - axis and the magnitude then is equal to this x component. To attain this, the x and y registers iteratively add or subtract the shifted values of y and x respectively so that the y register dies down to zero.

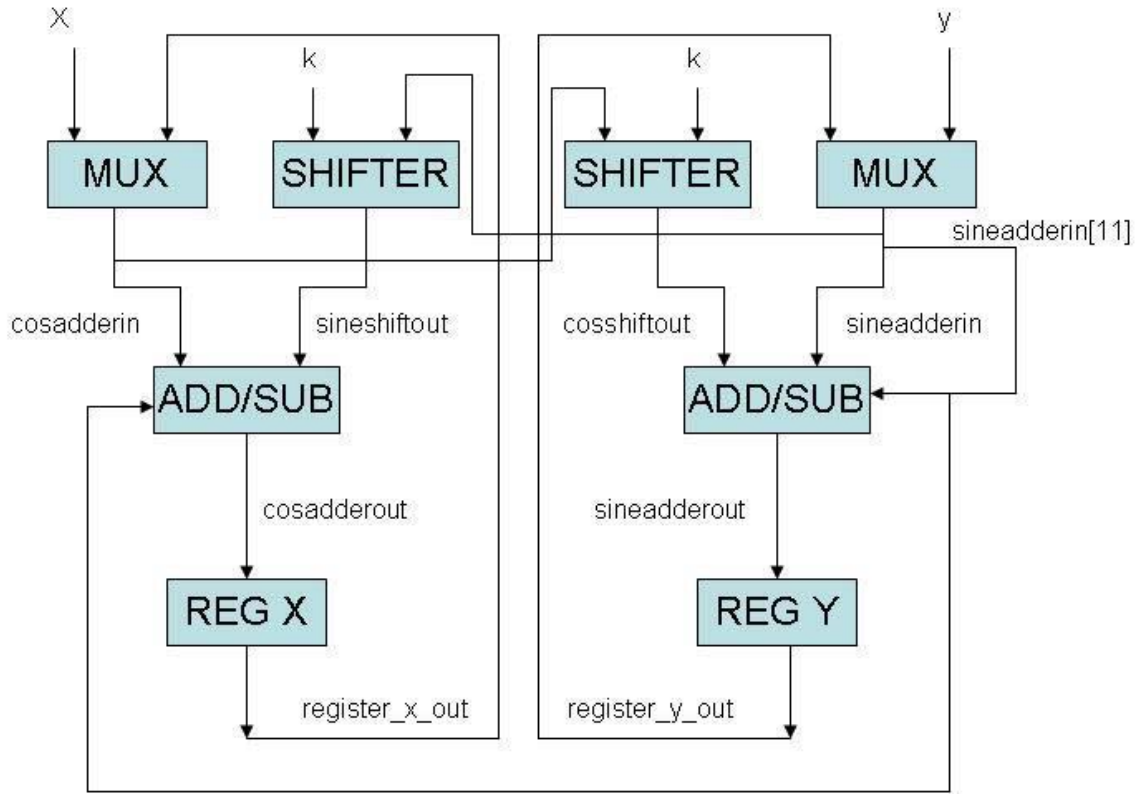


Fig 3.2.1: Block Diagram, Vectoring CORDIC

At the start of the iterations, we need to bring the vector in the region $+90$ to -90 degrees. The original vector $x + jy$, if in the 1st or the 2nd quadrant, is rotated by -90 degrees to get it in the region $+90$ degrees to -90 degrees. If the vector $x + jy$ is in the 3rd or the 4th quadrant, it is rotated by $+90$ degrees to get it in region $+90$ degrees to -90 degrees.

3.3 Log – Shifter

The shifter used is a Log –Shifter that shifts a 12 bit number c_in arithmetically by k bits to the right. It uses the combination of 3 modules, namely single, double, triple that respectively shift the number by $2^0 = 1$, $2^1 = 2$, $2^2 = 4$ bits to the right. For example, when $k = 101$, $k[0] = 1$ causes the 'single' module to shift c_in by 1 bit to the right. This is fed as input to the 'double' module. As the control input of this block, $k[1] = 0$, this module passes its input without any shifts to the 'triple' module. As the control input of this block, $k[2] = 1$, this module shifts its input by 4 bits to the right. So the input is cumulatively shifted to the right arithmetically by $4 + 1$ bits = 5 bits to the right. ($k = 3'b101 = 5$)

The block diagram is as shown in Fig. 3.3.1.

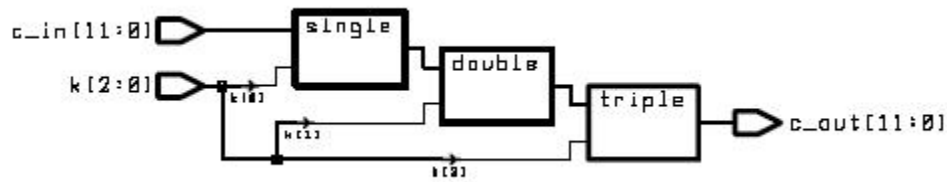


Fig. 3.3.1: Log – Shifter

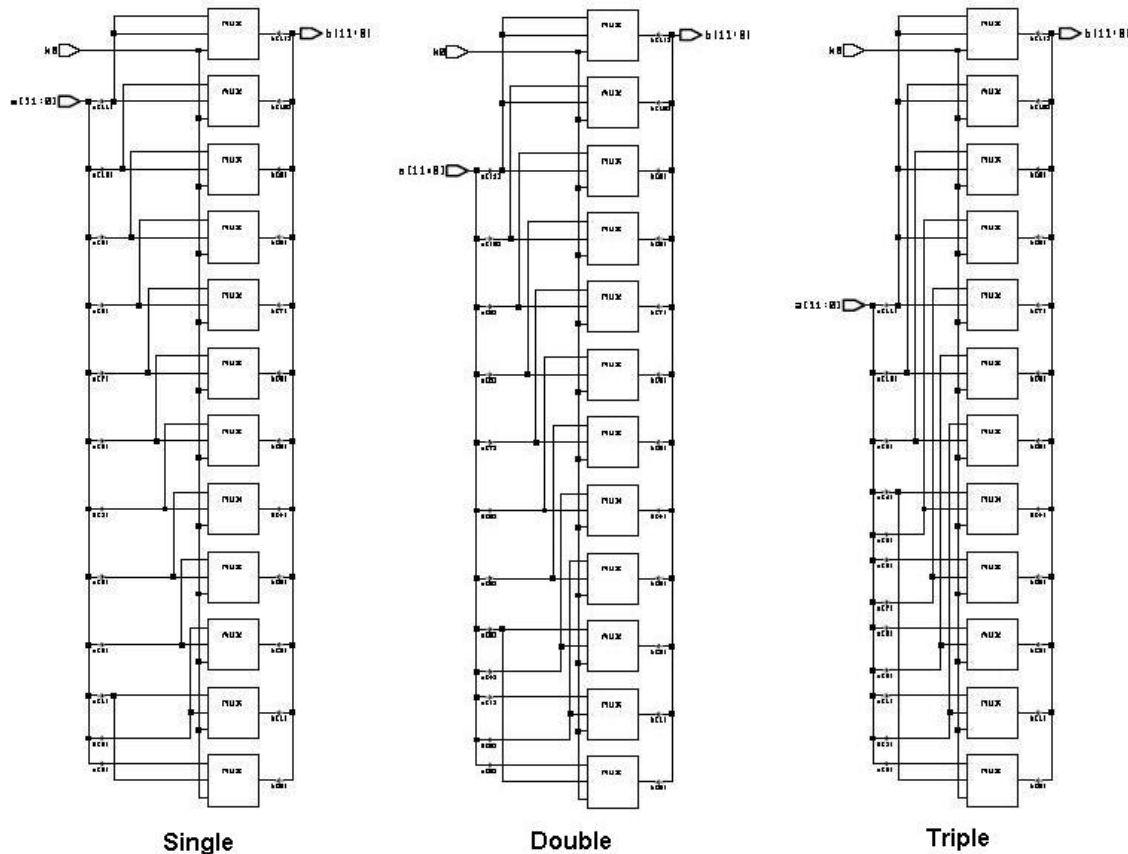


Fig. 3.3.2: The three components: Single, Double, Triple.

3.4 Look-Up Table

The LUT is required by the angle accumulator register in the rotation cordic module to compute the angle values that need to be added or subtracted from the accumulated angle.

The angle values stored are stored in the following format:

->12 bit representation is used.

->The MSB, bit[11], has a binary weight of -180 degrees.

->The next bit, bit[10] has a binary weight of +90 degrees.

->Then, the next successive bits, bit[9:0] have binary weights of $+90/(2^n)$ where n varies from 1 for bit[9] to 10 for bit [0]. Thus the least count or the least angle that can be represented in this system is $90/(2^{10}) = 0.087890625$ degrees.

In our code, the LUT is made using a simple case statement as shown in the following code segment. (Fig. 3.4.1)

```

module lookuptable(count_out,angle);
input [2:0] count_out;
output [11:0] angle;
reg [11:0] angle;

always @ ( count_out )
case (count_out)
3'b000:angle=12'b0010_0000_0000;//0 45 45
3'b001:angle=12'b0001_0010_1110;//1 26.54296875 26.56505
3'b010:angle=12'b0000_1010_0000;//2 14.0625 14.03624
3'b011:angle=12'b0000_0101_0001;//3 7.119140625 7.12502
3'b100:angle=12'b0000_0010_1001;//4 3.603515625 3.57633
3'b101:angle=12'b0000_0001_0100;//5 1.7578125 1.78661
3'b110:angle=12'b0000_0000_1010;//6 0.87890625 0.89517
3'b111:angle=12'b0000_0000_0101;//7 0.439453125 0.44761
endcase

endmodule

```

Fig.3.4.1: LUT Code

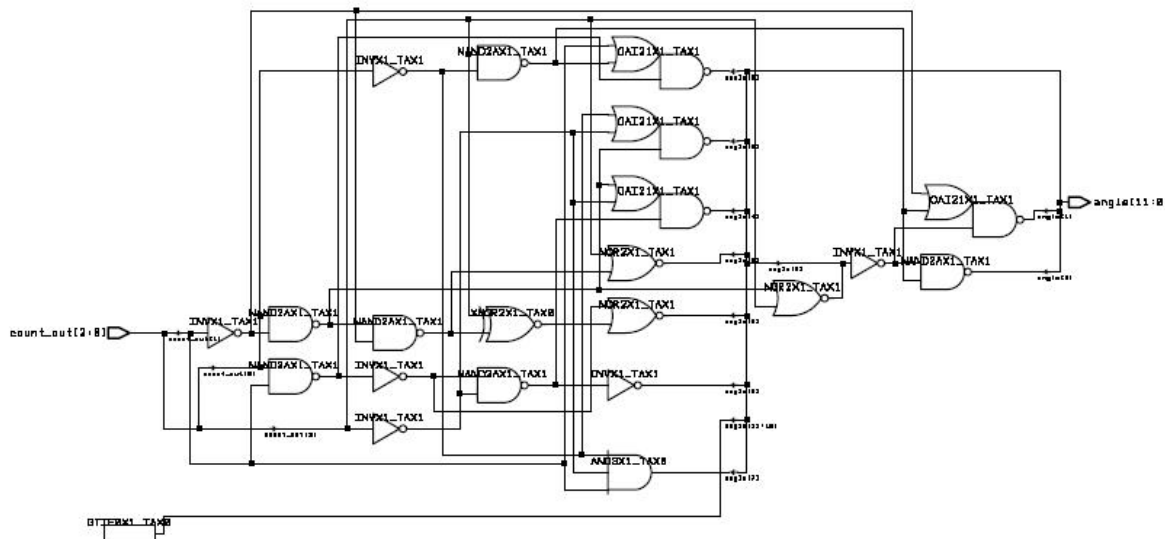


Fig. 3.4.2: LUT Circuit

Now, the angles stored in the LUT describe the angle to be used at each of the 8 iterations. These angles are calculated as follows:

$$\text{Angle at iteration } i \ (i = 0, \dots, 7) = \text{atan}(K) \text{ where } K = 2^{(-i)}.$$

Thus, at first iteration, i.e. when count_out = 3'b000, angle = 45 degrees. The representations we have used and the actual angles that are to be used are shown alongside in the code at the relevant place. Please note that we have used the best representation possible to minimize error. The angles differ by at max the least count of 0.087890625 degrees.

3.5 12-bit Full Adder

We use the fulladder module to make a 'cascaded' 12 bit adder block. As explained for the fulladder, this adder works as carry bypass.

This 12 bit adder is used as an adder/subtractor for two 12 bit numbers: a & b

The addition subtraction depends on the sign bit. sign = 0 means addition, sign = 1 means subtraction. For subtraction, using an EXOR inverter array, the 1's complement of b is passed to the cascade of 3 fulladder blocks along with making the input sign = 1. This is evident from the block diagram as shown in Fig. 3.5.1.

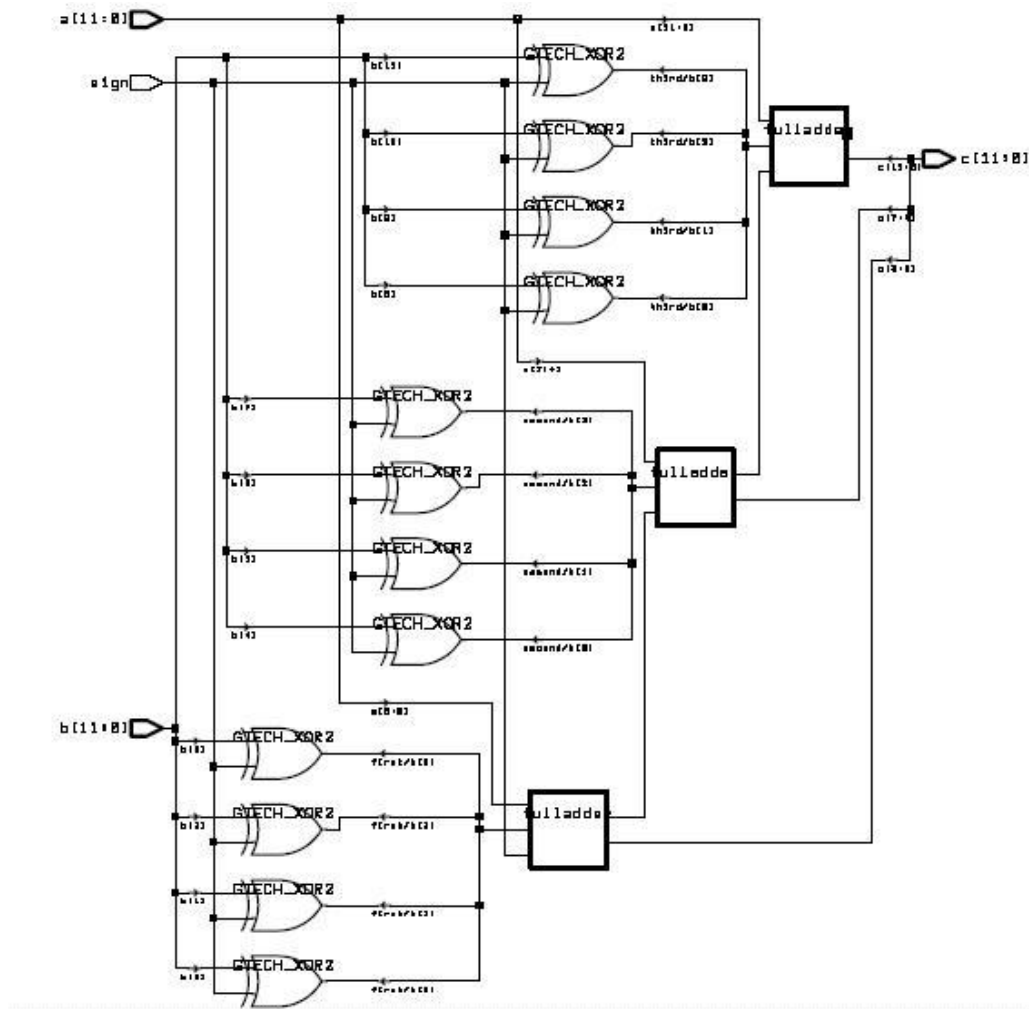


Fig. 3.5.1: Block Diagram, 12 Bit Full Adder

3.5.1 4 Bit Full Adder

The inputs are two 4 bit numbers, a and b and a single bit carry, c_in. The output is 4 bit c and single bit carry out c_out. The method used is that of carry propagate generate. When cascaded in ripple form with other similar blocks to make a 12 bit fulladder, carry bypass takes place as follows:

The (p3 & p2 & p1 & p0 & c_in) minterm sees to it that when all of p3, p2, p1 & p0 are HIGH, c_out = c_in, thus bypassing all the internal carries generated.

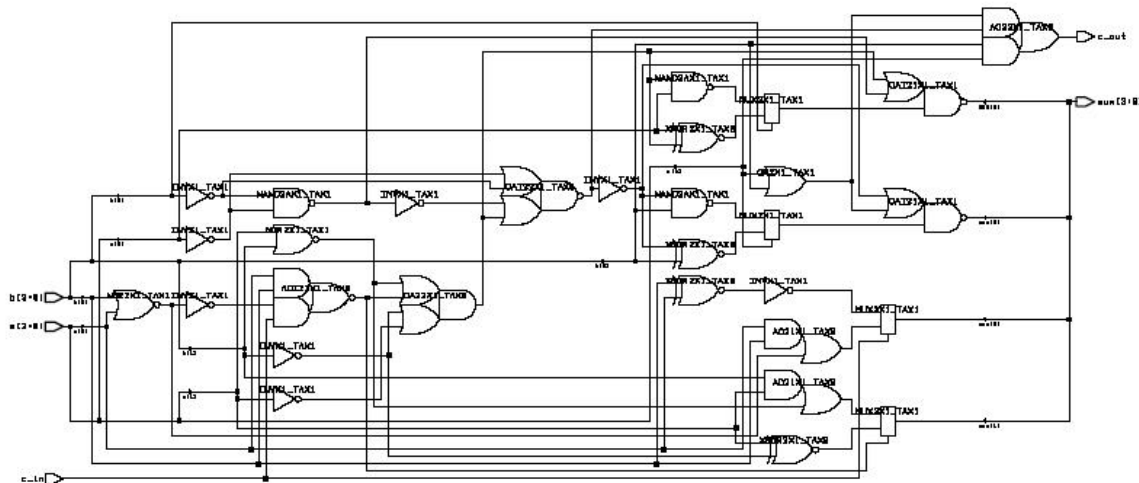


Fig.3.5.1.1: Block Diagram, 4 bit Full Adder

4. Comparison of Results with MATLAB 6.5[®]

The Verilog code was checked on the Cadence Verilog Simulator and the output values were computed for 5 different sets of input vectors. These results were then compared with the outputs obtained from Matlab.

It is observed that in all cases, the average error in the computation is 1.2 %. Such an error percentage is good enough for ordinary image analysis. However in case of Biomedical Image Processing, a lower error percentage will be required to prevent errors in decision making.

This error in computation can be attributed to the fact that we have used integer representation for the pixels. But while computing the error percentage we have considered fractional parts also. Also, we have used 8 bit representation of the numbers. This reduces the precision. The error percentage will come down if we use a higher bit representation and more number of iterations in CORDIC algorithm.

The .m file and the output array is shown in the Fig. 4.1 below.

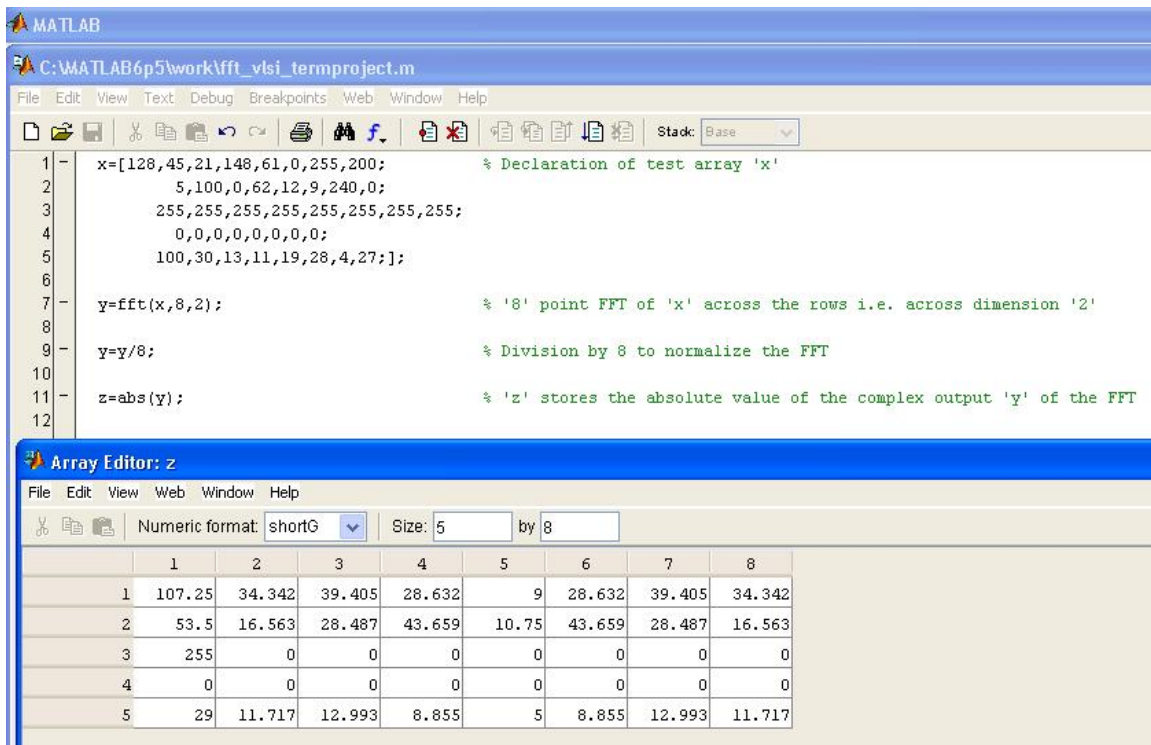


Fig. 4.1: Computation of FFT using Matlab

The comparison results are then tabulated as follows:

Sample # 1

Input Vector	128	45	21	148	61	0	255	200
Verilog O/P	183	57	64	47	15	47	64	57
÷ 1.647	111.11	34.6	38.86	28.54	9.11	28.54	38.86	34.6
MATLAB O/P	107.3	34.342	39.405	28.632	9	28.632	39.405	34.342

Sample # 2

Input Vector	5	100	0	62	12	9	240	0
Verilog O/P	92	27	47	71	17	71	47	27
÷ 1.647	55.86	16.39	28.54	43.11	10.32	43.11	28.54	16.39
MATLAB O/P	53.5	16.563	28.487	43.659	10.75	43.659	28.487	16.563

Sample # 3

Input Vector	255	255	255	255	255	255	255	255
Verilog O/P	4039	(Range Overshoots – Erroneous Output)						
÷ 1.647	(Not Applicable)							
MATLAB O/P	255	0	0	0	0	0	0	0

Sample # 4

Input Vector	0	0	0	0	0	0	0	0
Verilog O/P	0	0	0	0	0	0	0	0
÷ 1.647	0	0	0	0	0	0	0	0
MATLAB O/P	0	0	0	0	0	0	0	0

Sample # 5

Input Vector	100	30	13	11	19	28	4	27
Verilog O/P	48	19	21	15	8	15	21	19
÷ 1.647	29.15	11.54	12.75	9.11	4.86	9.11	12.75	11.54
MATLAB O/P	29	11.717	12.993	8.855	5	8.855	12.993	11.717

5. Future Work

5.1 Further Improvement in Architecture.

One way in which the present implementation can be improved is by changing the input output process. The input output block remains idle when processing is going on. We cannot enter new sets of data as long as the entered set has been completely computed. The new proposed architectural modification takes care of the fact that when computation of one is going on, input and output blocks are not staying idle. This will lead to kind of pipelined input output architecture for the whole block.

ALTERNATIVE INPUT OUTPUT SCHEME

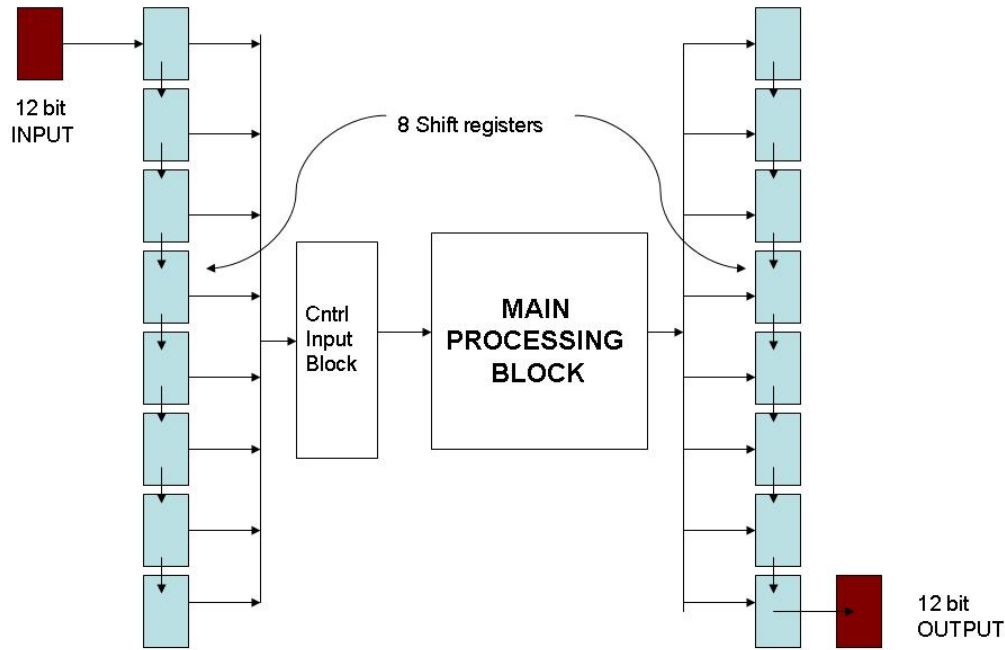


Fig. 5.1.1: Suggested Improvement in Input – Output Architecture

8 bits are entered serially into the 8 shift registers. After the 8 clock pulses only the 8 sets of numbers are entered to the block for actual processing. We know that the processing will require 12 clock pulses more. This time is utilized to enter new sets of data into the shift registers. Similarly, previously computed sets of data after the VECTORING CORDIC can be equivalently shifted out. This will give rise to additional hardware but there will be considerable improvement in the time complexity.

5.2 Interfacing with DSP kit.

With the availability of a DSP kit, the FPGA can be interfaced with a computer. An image stored on the computer can then be converted into a digital bit stream that can feed to our FFT block. The output can then be reconverted to the Fast Fourier Transformed Image.

5.3 As a basic block in other Image Transformation Techniques.

The present FFT block can be used as a major computational block in various other transforms like the Radon Transform.

References:

- [1] J. G. Proakis and D.G. Manolakis, “Digital Signal Processing, Principles, Algorithms and Applications.” 3rd Edition, 1998, Prentice Hall India Publications.

- [2] B. Das and S. Banerjee, “Some Studies on VLSI Based Signal Processing for Biomedical Applications.” Ph.D. Thesis.

- [3] Ray Andraka, “A survey of CORDIC Algorithms for FPGA based computers”. Proceedings of the 1998 ACM/SIGDA sixth International Symposium on Field Programmable Gate Array.

Web References:

<http://www.dspguru.com/info/faqs/cordic.htm>