ASIC IMPLEMENTATION OF FFT ENGINE FOR AUDIO DRIVER

A thesis submitted in the partial fulfilment of the requirements of the Degree of

Master of Technology
in

VLSI DESIGN AND EMBEDDED SYSTEMs

Submitted by

Ashutosh Kumar Singh
(Roll No: 213EC2212)

Department of Electronics and Communication Engineering
National Institute of Technology Rourkela
Rourkela - 769 008, India
May 2015

ASIC IMPLEMENTATION OF FFT ENGINE FOR AUDIO DRIVER

A thesis submitted in the partial fulfilment of the requirements of the Degree of

Master of Technology
in

VLSI DESIGN AND EMBEDDED SYSTEMs

Submitted by

Ashutosh Kumar Singh
(Roll No: 213EC2212)

Under the guidance of
Prof. Debiprasad Priyabrata Acharya

Department of Electronics and Communication Engineering
National Institute of Technology Rourkela
Rourkela - 769 008, India
May 2015

Department of Electronics & Communication Engineering
NATIONAL INSTITUTE OF TECHNOLOGY, ROURKELA
ODISHA, INDIA – 769 008

CERTIFICATE

This is to certify that the thesis titled **"ASIC IMPLEMENTATION OF FFT ENGINE FOR AUDIO DRIVER"** submitted to the National Institute of Technology, Rourkela by **Ashutosh Kumar Singh**, Roll No. **213EC2212** for the award of the degree of **Master of Technology** in **Electronics & Communication Engineering** with specialization in **"VLSI Design and Embedded Systems"**, is a bonafide record of research work carried out by him under my supervision and guidance. The candidate has fulfilled all the prescribed requirements.

The thesis, which is based on candidate's own work, neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere. To the best of my knowledge, the thesis is of standard required for the award of the degree of **Master of Technology** in Electronics & Communication Engineering.

Place: Rourkela

Prof. Debiprasad Priyabrata Acharya
Department of Electronics &
Communication Engineering
National Institute of Technology
Rourkela-769 008 (INDIA)

Department of Electronics & Communication Engineering
NATIONAL INSTITUTE OF TECHNOLOGY, ROURKELA
ODISHA, INDIA – 769 008

DECLARATION

I certify that

a) The work contained in the thesis is original and has been done by myself under the supervision of Prof. D. P. Acharya, Department of Electronics and Communication Engineering, NIT Rourkela, and Er. Ashvin Kumar G Katakwar from Sankalp Semiconductor Pvt. Ltd (Kolkata).
b) The project work written in the thesis is a part of my internship that I have completed at Sankalp Semiconductor Pvt Ltd (Kolkata).
c) The work has not been submitted to any other Institute for any degree or diploma.
d) I have followed the guidelines provided by the Institute and Company in writing the thesis.

Ashutosh Kumar Singh

1st Jan 2015

DEDICATED

TO

MY PARENTS AND FRIENDS

Acknowledgement

It is my immense pleasure to avail this opportunity to express my gratitude and regards to my project guide **Prof. D. P. Acharya,** Department of Electronics and Communication Engineering, NIT Rourkela for his valuable advice and support throughout my project work. I am especially indebted to him for teaching me both research and writing skills, which have been proven beneficial for my current research and future career. Without his endless efforts, knowledge and patience, this research would have never been possible.

I express my sincere gratitude to **Prof. K. K. Mahapatra, Prof. P. K. Tiwari, Prof. A. K. Swain, Prof. M. N. Islam** and **Prof. Santanu Sarkar**, for their support, feedback and guidance throughout my M. Tech course duration. I would also like to thank all the faculty and staff of the ECE department, NIT Rourkela for their support and help during the two years of my student life in the department.

I would also like to thank Mr. Prajeet Nandi, and Ashvin Kumar G Katakwar, Dhiraj Kumar, Hirak Talukdar, chandrima chaudhari, Rajsekhar Sinha and Arpan Gupta from Sankalp Semiconductor Pvt. Ltd. for their valuable advice and support throughout my internship. I am grateful to Saragadam Sailaja for her priceless support during our internship at Kolkata.

I must express my deep appreciation and gratitude to PhD scholars Mr. Umakanta Nanda and Mr. Debasish Nayak who were always ready to share their knowledge throughout my course. I also extend my gratitude to my lab-mate Sarika Anil Kumar and Naresh Thakur for the worthy ideas we had shared on our respective research areas. I am really thankful especially Santosh Padhy, Nishchay Malik, Nitin Jain, Anil Rajput, Mukesh Kumar Kushwaha and Chandan Maurya for being my guardian angel and always standing with me during my stay at NIT. I also extend my whole-hearted gratitude to one and all my batch mates and other friends for their immense cooperation without whom my stay in NIT would not have been so enjoyable and memorable.

Last but not least I thank my family whose constant support and encouragement, always help me to move forward in life even during hard times.

Ashutosh Kumar Singh

Chapter-1

Introduction

1.1 Objective

Dynamic performance of an audio driver is measured in using using the parameters SNDR (Signal to Noise plus Distortion Ratio), SFDR (Spurious Free Dynamic Range), THD (Total Harmonic Distortion), SNR (Signal to Noise Ratio) and DC component in the signal.

To improve the dynamic performance of audio driver system, we need to monitor the these performance parameters of the signal. We can design a system that accepts analog audio signal as input, converts it into a digital signal using an ADC, calculates its various performance parameters and feed back to a control block. Control block takes these performance parameters and compares it with the stored reference values. If control bloc finds that any parameter is going away from given limited value then it tries to offset that parameter by sending appropriate signal to corresponding block.

We can take an example of DC offset in signal. Suppose desired DC component in the audio signal is zero that is reference value for DC component. Now if measured DC component is 0.5 mV then control can send feedback signal to offset control block to subtract 0.5 mV from each sample and after subtracting this offset from each sample, if we will take the FFT of input samples then measured DC in the analog signal will be zero.

Since we can not subtract each measured value from the samples so we have to decide the fixed step of increment from the least possible value to max possible value. In the same way, we can develop the technique to offset other parameters also.

Block diagram shown in Figure 1.1 clearly represents the motivation behind the thesis work. ADC is sampling the analog signal from mixed signal system and converting it into digital format. N such samples are stored in memory to compute its FFT and half of the FFT bins are stored in memory to compute the various parameters. Control block is performs the comparison and sends the

appropriate feed back signal to mixed signal system.

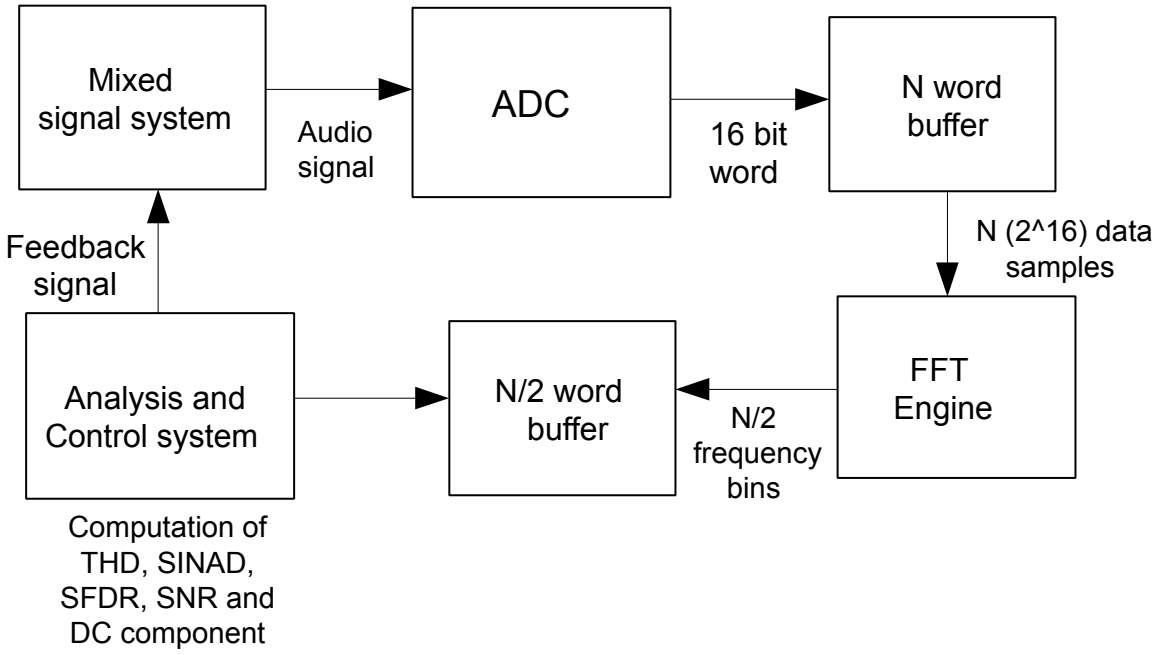

Figure 1.1 Block diagram of FFT engine

We can compute these parameters using FFT plot of audio signal. FFT Block will take digital input signal, so we need one Memory Block that will store the required input sample and after computing the FFT of input signal, we need an Analysis Block block that can compute these parameters. So finally the Top-level system consist of three subsystems.

1. Memory Block
2. FFT Block
3. Analysis Block

Since it is a digital system, so we can directly proceed for HDL codinging (either verilog or VHDL) but HDL modeling may have difficulties in tracking the result at each step. As we can take the example of HDL modeling of 65536 point FFT engine, it is difficult to provide 65536 samples of input

sine wave and to track the corresponding 65536 output samples. Again if we want to introduce some DC offset in the input sine wave, then we need to change the whole input array.

Another way to implement the system is using MATLAB. We can easily write the behavioral level description of an algorithm in MATLAB and select an appropriate algorithm for FFT Block and Analysis Block and we can very easily track the corresponding result by plotting it. Once behavioral level modeling is done, we can proceed for hardware level modeling in Simulink. At this level of modeling, we can replace the MATLAB programs for each block with corresponding FSM and required hardware (processing elements like adders, subtractors, multipliers etc). We can verify the entire model at each step of progress.

Next step is to convert the entire Floating point model into fixed point model. Fixed point model is more closer to a RTL model. At this step, again we can verify the model and track the requirements.

Next step is to generate Verilog code using designed Simulink model and to verify the generated code. To design the entire digital system, we have followed this design flow.

Our main goal is to reduce the hardware requirement in the design. In this design, speed of operation has a lower priority because in audio driver system after getting the required number of sample, processing will not affect the operation of remaining system. So we can process the data with a lower speed without affecting the overall performance of chip.

1.2 Problem description

Block diagram shown in Figure 1.1 clearly represents the blocks needs to be designed for the appropriate functioning of the feedback loop. Initially to ensure the proper functioning of the system, we can make a behavioral level model in MATLAB for each and every block. At this level of modeling, we can decide the appropriate algorithm that meets our requirements. So at behavioral level of modeling we have design following blocks

1. Modeling of ADC in MATLAB
2. Modeling of FFT engine in MATLAB
3. Modeling of Analysis block in MATLAB

Initially these three blocks can be designed in MATLAB. In Modeling of ADC, we just need to design the quantizer of ADC because other blocks are working on floating point number system that can not accept the encoder output directly as their input. If we will use encoder then we need to use a DAC Digital to Analog Converter) so that output of ADC can be directly used by other blocks.

Since audio signals are low frequency signals (less than 20 kHz) and generally signals are over-sampled. In order to get a frequency resolution of less than 100 Hz for a sampling frequency range of 4 MHz to 6.5MHz, we have to compute 65536 (2^{16}) point FFT .

For audio driver application, our analysis range should be in audio range that is 20 Hz to 20 kHz. So while designing of Analysis block we need to concentrate only on this range of frequency. Any spur or harmonic outside this range should not be included in the analysis.

1.3 Thesis organization

The thesis consist of total 7 chapter including introduction as the first chapter. Second chapter focuses on basics of FFT, selection of FFT algorithm and behavioral level modeling of overall system in MATLAB.

Third chapter focuses on the implementation of implementation of top level system in Simulink. In this chapter we will decide the architecture of FFT engine, and its implementation using the available high-level resources (adders, subtractors, multipliers, shifter, multiplexers and memories).

Fourth chapter focuses on CORDIC algorithm and its importance in implementation. CORDIC algorithm has been used for the implementation of CORDIC multiplier, Absolute block and for the hardware implementation of Logarithmic function.

Fifth chapter focuses on Fixed point conversion of Simulink model. After fixed point conversion, the model behaves as a fixed point digital design as input and output data width of each hardware is fixed.

Sixth chapter focuses on generation of HDL code (Verilog) using MATLAB HDL coder and simulation of generated code using ModelSim.

Last chapter focuses on conclusion, scope of improvement, and future work.

Chapter-2

Behavioral level modeling of FFT Engine

2.1 Importance of Discrete Fourier Transform

If any signal x(t) is aperiodic and continuous in nature then Fourier Transform of the signal will also be aperiodic and continuous. If it is discrete and aperiodic then also corresponding frequency domain signal will be continuous in nature.

Only in case of Discrete Fourier Transform (DFT) signal is discrete in nature in both time domain as well as frequency domain. There is an inherent advantage of discrete time domain and frequency domain signal that it can be stored and processed using a digital computer and we can characterized and analyzed the signal in frequency domain.

DFT of N samples of a discrete signal x(n) is given by Equation (1.1) where X(k) is DFT of x(n) and k is frequency domain index.

$$X[k] = \sum_{n=0}^{n=N-1} x(n) W_N^{nk} \tag{2.1}$$

where $W_N = e^{-j*2pi/N}$ and it is called twiddle factor.

2.2 Computation complexity in DFT

For N point DFT, it requires N^2 complex multiplications where each complex multiplication uses 4 real multiplications and two real addition.

It requires N*(N-1) complex additions where each complex addition requires two real additions. So total $4*N^2$ Real multiplications and $2*N^2 + 2*N*(N-1)$ real additions.

2.3 Available FFT algorithms to reduce the computational complexity of DFT

There are so many algorithm that can reduce the computational complexity of DFT, notably

1. Radix-2 Algorithm
2. Radix-4 Algorithm
3. Split Radix algorithm
4. Fast Hartley Transform

Radix-2 FFT algorithm is the basic algorithm to compute DFT with lesser computational complexity. For N point DFT, FFT algorithm needs

- $(N/2)*\log_2 N$ complex multiplications
- $N*\log_2 N*$ complex additions

Since we need to implement a FFT engine for 65536 points and due to hardware constraint, we can not go for parallel implementation of Radix-2 FFT algorithm. To compute the N point FFT using a single butterfly, we have to reuse it for $(N/2)*\log_2 N$ times. So FFT computation may be very much slower for 65536 points.

Radix-4 FFT algorithm may reduce the computation time with little increase in hardware (to implement CORDIC butterfly). For N point DFT, this FFT algorithm needs

- (N/4)*log4N complex multiplications
- $N*\log_4 N*$ complex additions

So radix-4 algorithm is more efficient than radix-2 algorithm in terms of computation complexity. To compute the N point FFT using a single butterfly, we have to reuse it for $(N/4)*\log_4 N$ times. So radix-4 algorithm will be faster for FFT computation using single butterfly.

Split radix algorithm have lesser complexity than Radix-2, and Radix -4 algorithm but it is

difficult to design a FSM for the rotation of data since FFT structure is not planer and there is requirement of both radix-2 and radix-4 butterfly in processing.

The computation complexity of radix-2 FHT is same as radix-2 FFT for real data but for complex data it increases by a factor of 2.

So finally, Radix-4 FFT algorithm is most suitable algorithm to compute the FFT for 65536 points and using Single butterfly and one Finite state machine.

2.4 Behavioral level modeling and simulation of FFT algorithm in MATLAB

for the modeling of N point radix-4 FFT algorithm , a few observations are required as

1. Each stage has the same number of butterflies (number of butterflies = N/4, N is number of points)
2. The number of DFT groups per stage is equal to ($N/4^{stage}$)
3. The difference between the upper and lower leg is equal to $4^{stage-1}$
4. The number of butterflies in the group is equal to $4^{stage-1}$

2.5 Step of implementation for N point FFT

1. Store 'N' samples in a buffer
2. Get the length of sequence
3. Zero padding to make it N=2^n point sequence where n is an integer
4. Rearrangement of data in proper sequence (Input should be in bit reversed order to get the output in normal order)
5. Apply the FFT algorithm to compute DFT
6. Store first N/2 real and imaginary points to get the frequency domain sequence
7. Compute absolute value of stored N/2 complex points and store computed N/2 points for further

processing

2.6 MATLAB simulation of FFT algorithm

MATLAB algorithm has been checked for for a signal x(t) where

x(t)=a*cos(2*pi*fm*t)+h*cos(2*pi*l*fm*t)+nv*rand(N)

This signal consist of cosine signal with frequency **'fm'** and peak amplitude of **'a'**, its **l^{th}** harmonic with peak amplitude of **'h'** and a random noise signal with standard deviation **'nv'**. **'N'** is the number of points in FFT and **'t'** is the sample parameter.

Algorithm has been simulated for the specifications shown in Table 2.1

fm (Hz)	fs	l	a	h	nv	N
1000	5000	2	1	0.5	0.4	4096

Table 2.1 Simulation parameters for input signal

For the given signal, magnitude and phase plots are plotted using the written algorithm and results are checked against inbuilt MATLAB function for FFT.

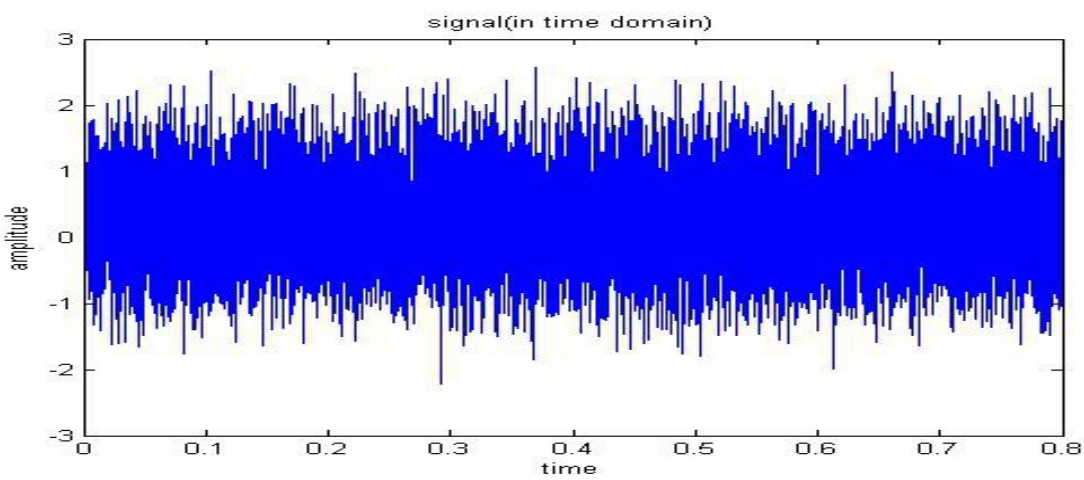

Figure 2.1 Input signal for simulation

Since phase information is not required in computation desired parameters, so a part of algorithm that computes the phase information can be removed in next step of implementation. Figure 2.1 shows the input signal in time domain and Figure 2.2 is shows the magnitude and phase plot of computed DFT.

2.7 MATLAB functions used in FFT Algorithm

FFT algorithm written in MATLAB is using some MATLAB functions that can not be directly

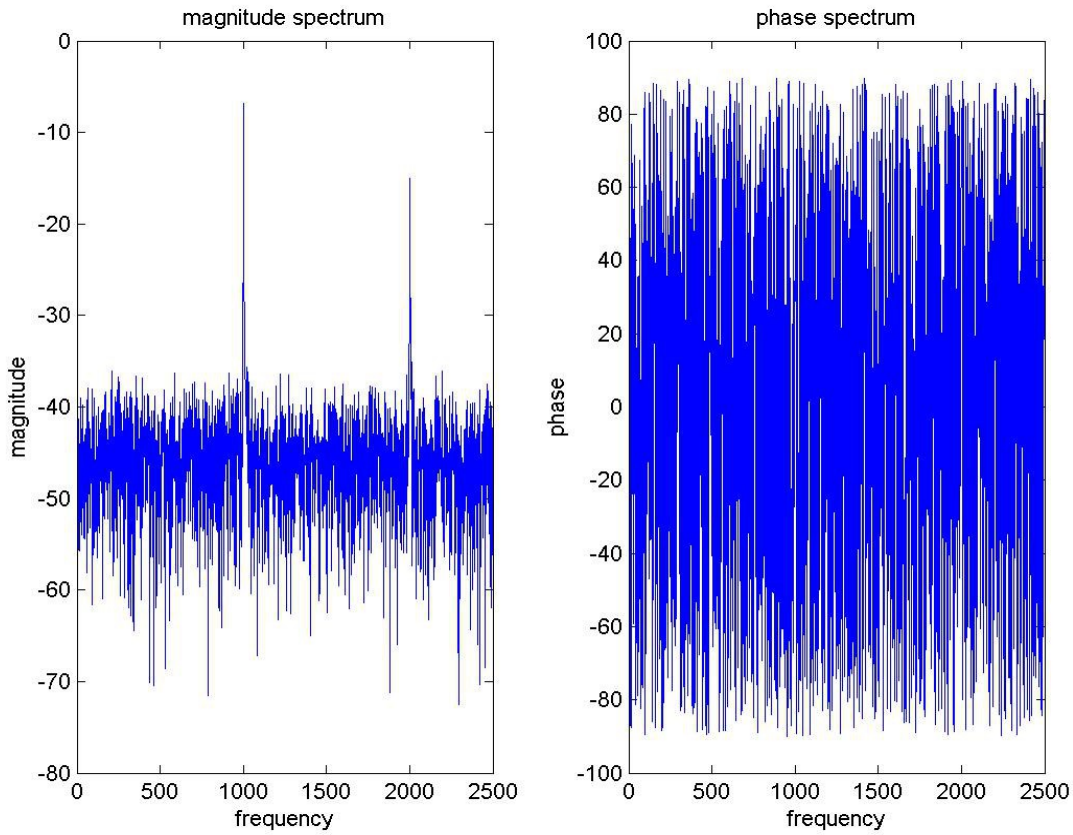

Figure 2.2 Magnitude and phase plots for the computed FFT

synthesized using available high level resources (Adders, Subtractors, Multipliers, Multiplexers, shifters, Memories). So in next level of implementation, these functions are either written in

synthesizable form or a separate hardware section is designed to implement such functions (such as Logarithmic and absolute functions). These functions are listed bellow.

1. x=bit_plane_reverse(x, n)
2. x=bit_plane_reverse_sequence(x)
3. x=absolute(x1, x2, n)
4. y= phase1(x1, x2, n)
5. n1=next_pow_4(n)
6. z=mod1(x, y)
7. x=ceil1(x)
8. y=floor1(y)
9. n=dec_2_bin(x)

First two functions are used to rearrange the index of input data in bit reversed order. 'absolute' function is used to compute the absolute value of a complex number. 'phase1' function is used to extract the phase information from computed DFT. Remaining functions are also used in rearranging of sequence.

2.8 MATLAB Algorithm for Analysis Block

Analysis block computes different parameters that characterize the dynamic performance of input signal using the computed DFT. It will compute

1. Total Harmonic Distortion (THD)
2. Signal to Noise plus Distortion Ratio (SNDR)
3. Spurious Free Dynamic Range (SFDR)
4. Percent THD plus Noise

5. Signal to Noise Ratio (SNR)
6. DC Component

2.8.1 Total Harmonic Distortion (THD)

It is given by the ratio of the root mean square (rms) value of the fundamental signal to the average value of the root-sum-square (rss) of its harmonics.

If Fm is frequency of harmonic then its corresponding frequency bin will be (Fm*N)/Fs, where 'N' is number of FFT points and 'Fs' is sampling frequency. Analysis Block algorithm is using 3 three user defined MATLAB functions to compute the Total Harmonic Distortion.

1. Fe= getmax(Fb) : To get the frequency bin with peak amplitude near ($\pm$ 5 bins) expected frequency bin 'Fb'
2. [Fu,Fl]=getrange(Fe) : To get the spreading range of harmonic
3. Powh=get_pow(Fe,Fu,Fl) : To get the exact power of this harmonic

Algorithm first computes the frequency bin corresponding to harmonic frequency and assumes that harmonic power lies in this frequency bin. To ensure this, algorithm uses 'getmax' function and checks nearby 5 bins in both side of assumed frequency bin. If in this range, any bin has higher amplitude than assumed bin then that frequency bin is considered as peak frequency bin in which corresponding harmonic power lies.

Due to spectral leakage, total power spreads into nearby frequency bins. To consider this power leakage, algorithm uses 'getrange' function. It computes the spreading of harmonic bins by considering the peak bin as reference.

'get_pow' function simply computes the power of each harmonic by computing the square sum

of bins from its lower range to its upper range.

Since algorithm is considering only half of the FFT bins for the computation of harmonic power, so the computed harmonic power is half of the total harmonic power. Finally algorithm multiplies the power of each harmonic by two to compute the correct results.

2.8.2 Signal to Noise plus Distortion Ratio (SNDR)

It is the ratio of the rms value of fundamental signal amplitude to the average value of the rss of all other components, including harmonics components, but excluding DC component.

Algorithm first computes the total power that lies in the signal excluding DC Component (amplitude of the first bin). In next step it subtracts the signal power to compute total noise plus distortion power.

2.8.3 Spurious Free Dynamic Range (SFDR)

It is the ratio of worst spur that lies in first Nyquist zone to peak signal bin. Worst spur is searched from second bin to last bin excluding the range of fundamental signal.

2.8.4 Signal to Noise Ratio (SNR)

Signal-to-noise ratio is computed in the same way as SINAD, except that the harmonic components are excluded, and only noise terms are used in computation. Since we have already calculated the total noise plus distortion power and total harmonic power separately so we can subtract the harmonic power from that and it will give total noise power.

2.8.5 Percent THD plus Noise

It provides the same information as given by SNDR (related to total noise plus distortion). It is given by Equation (2.2)

$$Percent\ THD\ plus\ Noise = \frac{Noise\ plus\ Distortion\ power}{Signal\ power} * 100 \qquad (2.2)$$

2.8.6 DC Component

DC component in the signal is simply the amplitude of first bin in the DFT spectrum. First bin of real part of computed FFT (before the computation of absolute value) gives the DC component with proper sign.

2.9 MATLAB simulation of Analysis block algorithm

Analysis Block is simulated for a signal with following parameters

- Signal frequency 10.68572998 kHz
- Sampling frequency 100 kHz
- Signal is quantized by a 14 bit ADC
- 65536 point DFT is calculated using FFT Block
- 32768 bins are fed to Analysis Block to compute all the desired parameters

For 14 bit quantization, expected SNR is approximately -86.04 dB and Analysis Block results are almost matching with expected results as shown in Table 2.2

Parameters	THD	SNDR	SNR	SFDR	Percent THD plus Noise
Results	-115.7 dB	-85.93 dB	-85.93 dB	-114.5 dB	0.0051

Table 2.2 Simulation results for Analysis Block

Input sign wave is ideal so there no harmonics and THD is less than -115 dB. This results in

almost equal value of SNDR and SNR (no harmonic distortion).

2.10 Modeling of ADC in MATLAB

ADC is required to quantize the ideal input sine wave. We can accurately track the Analysis block results for fixed bit quantization as it introduces known quantization error into ideal input sine wave. If a 'n' bit ADC is quantizing the ideal input sine wave with peak amplitude of unity then expected SNR will be almost (6.02*n+1.76) dB. So for 14 bit ADC, SNR will be appropriately 86.04 dB. The value of SNR can be tracked to check the accuracy of the top level system. Components of ADC that have been designed on MATLAB are

1. Quantizer (to map the sampled value with one of the available quantization level)
2. Encoder (to convert the quantization level into the binary form)
3. DAC (It is not a part of ADC but it is designed to ensure that ADC encoding is correct and we are getting the sampled value back with encoded bits)

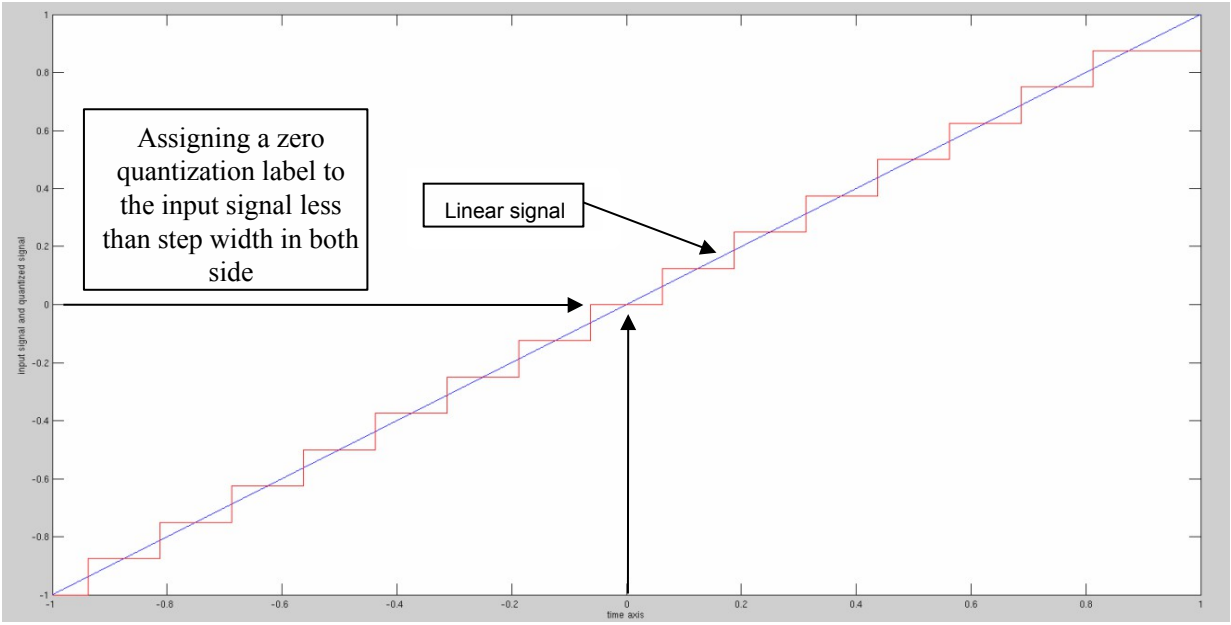

Figure 2.3 Characteristics of a mid-tread type quantizer designed in MATLAB

Quantizer is of mid-tread type, so if Quantizer is designed for a peak value of one, one has to keep the maximum amplitude of signal less than one minus half of the quantization width to keep the quantization error with in half of quantization width. Mean of the quantization error is zero for a mid trade type of quantizer. Figure 2.3 is showing the characteristic of mid-tread type quantizer designed in MATLAB.

Chapter-3

Modeling of FFT Engine in Simulink

3.1 Top level model of FFT Engine in Simulink

Next step of the project is modeling of FFT engine in Simulink using available high-level resources. All the functions used in algorithm must be implemented using dedicated hardware. At this level one can choose a suitable architecture that meets our requirement of hardware. Our aim is to minimize the requirement of hardware with optimum speed of operation.

Top level model integrates all the required blocks in a single test-bench. Behavioral level MATLAB algorithms are attached to corresponding Simulink blocks. Figure 3.1 shows the top level model of FFT engine.

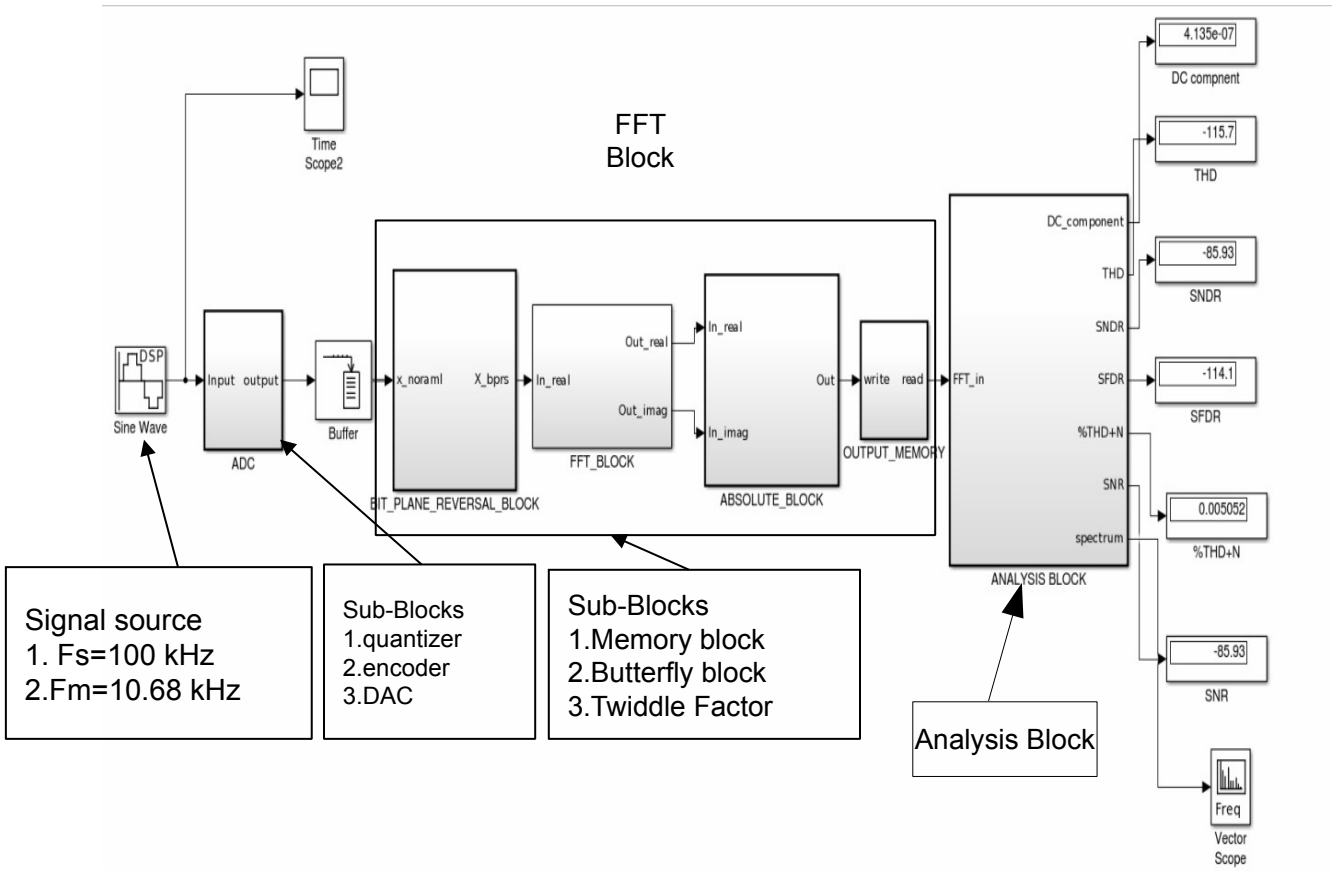

Figure 3.1 Top level model of FFT Engine

3.2 Different Blocks in FFT Engine

There are four different blocks in the top level model of FFT engine. Top level model has been designed in Simulink and it integrates all the blocks that will be extended next step of implementation.

1. Source Block

- Provides sampled signal with specified sampling frequency and signal frequency
- In this design, we are using coherent sampling and signal frequency is 10.68572998 kHz and sampling frequency is 100 kHz

2. ADC block

- It has three blocks, Quantizer, Encoder and DAC
- DAC is placed after ADC because currently designed algorithm is on Floating point number system that does not accept binary values
- In the next step of implementation when the Floating point model will be converted into Fixed point model then DAC will be removed
- In our design, we are using a 14 bit ADC

3. Buffer block

- Buffer block is used to take the samples from ADC after every Ts interval of time and stores N samples and produces a frame of N samples after every N*Ts interval of time
- Since our FFT block is designed for 65536 Points so the buffer length is also 65536

4. Bit Plane Reversal Block

- This block has two parts
- First part takes the index of sample as input and provides bit reversed index of this sequence as output

- Second block swaps the samples from those two indexes
- It does it for first half of the sequence except first sample since bit reversed indexes for the first and last index is itself that index

5. **FFT Block**

- It has three parts
- First part is memory block that has two parts. First part stores the real data sequence and second part stores the imaginary data sequence
- Since we are designing FFT block for real data so second part of memory is initialized to zero
- Second block is twiddle factor block that provides required twiddle factor in particular butterfly calculation
- Third block is butterfly block that takes the real and imaginary sequences from memory block and twiddle and takes required twiddle factor from twiddle factor block and performs the r point butterfly calculation

6. **Absolute Block**

- This block takes the input from real and imaginary part memory blocks and compute the absolute value of the complex sequence

7. **Output Memory**

- Stores N/2 points provided by absolute block

8. **Analysis Block**

- This block performs the analysis using stored N/2 points and calculates SNR, SINAD, SFDR, THD, DC Component, Percent THD plus Noise

9. **Top model also have**

- One time scope to see the input samples and one vector frequency scope to see the output

frequency spectrum of computed sequence

- Using set_param.m program we can change different block parameters like Fm, Fs, and N

3.3 Architecture of FFT block

There are three well known architecture are available for the implementation of FFT Block

1. Parallel Architecture
2. Pipelined Architecture
3. Memory based Architecture

Parallel architecture needs $(N/4)*\log_4 N$ radix-4 butterflies to compute 'N' point FFT using radix-4 FFT algorithm. This architecture is suitable for very high speed processing but hardware requirement is very high.

Pipelined architecture is generally used for real time signal processing which computes FFT in a sequential manner. This architecture needs $\log_4 N$ radix-4 butterflies to compute 'N' point FFT using radix-4 FFT algorithm. Hardware requirement in this architecture is less than Parallel architecture but this architecture is slower in comparison to Parallel architecture.

Memory based architecture is the slowest architecture among all the three architectures but it needs least hardware. This architecture computes the FFT using a single butterfly. A control logic block (a finite state machine) is used to manage the data to perform all the operations.

So finally the Memory based architecture is used for the implementation of FFT Block because it is the most hardware efficient algorithm for the implementation of FFT Block. Figure 3.2 is showing the block diagram of the architecture of FFT block. It is a memory based architecture. There are two RAMs and nine registers in this architecture. Registers holds the data before and after processing until it gets stored back to RAM. Control logic block is responsible for the rotation of data. Butterfly block is a radix-4 butterfly without CORDIC block. In this architecture it is assumed that twiddle factors are stored in ROM in Twiddle factor block and control logic logic is providing appropriate signal to fetch

the required sine and cosine value of the twiddle factor. In next step of implementation, twiddle factor block will be removed and CORDIC multipliers will be inserted in the in the Butterfly block to rotate the complex data with given angle. It will save the N word ROM used in the Twiddle factor block and real multipliers used in CORDIC block. CORDIC block will be discussed in the next chapter.

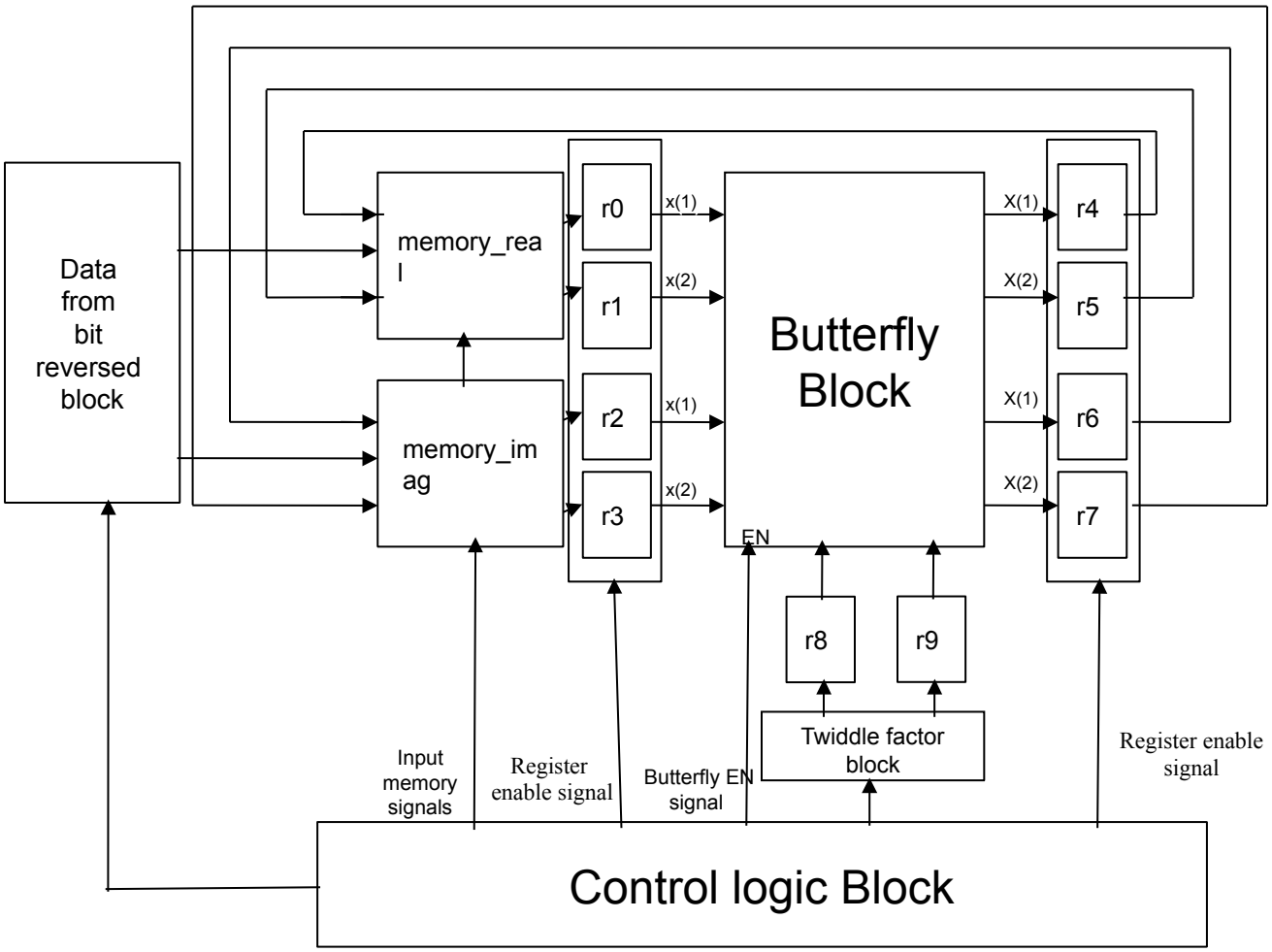

Figure 3.2 Block diagram of the architecture of FFT block

Architecture also consist of a bit index reversal block that rearranges the input data sequence into bit index reversed sequence to get the output in normal order.

3.4 Implementation of FFT block in Simulink

As per analysis and requirement, FFT Block will be implemented using Radix-4 algorithm

and Memory based architecture. Memory based architecture requires a single butterfly and a control logic block. Control logic block is a Finite State Machine that is responsible for the rotation of data.

3.4.1 Flow chart for the Control Logic Block

Figure 3.3 to Figure 3.6 is showing the flow chart for FSM used in FFT block. This flow chart is based on radix-2 algorithm and same procedure is followed for radix-4 algorithm except miner changes. These changes are discussed in next section.

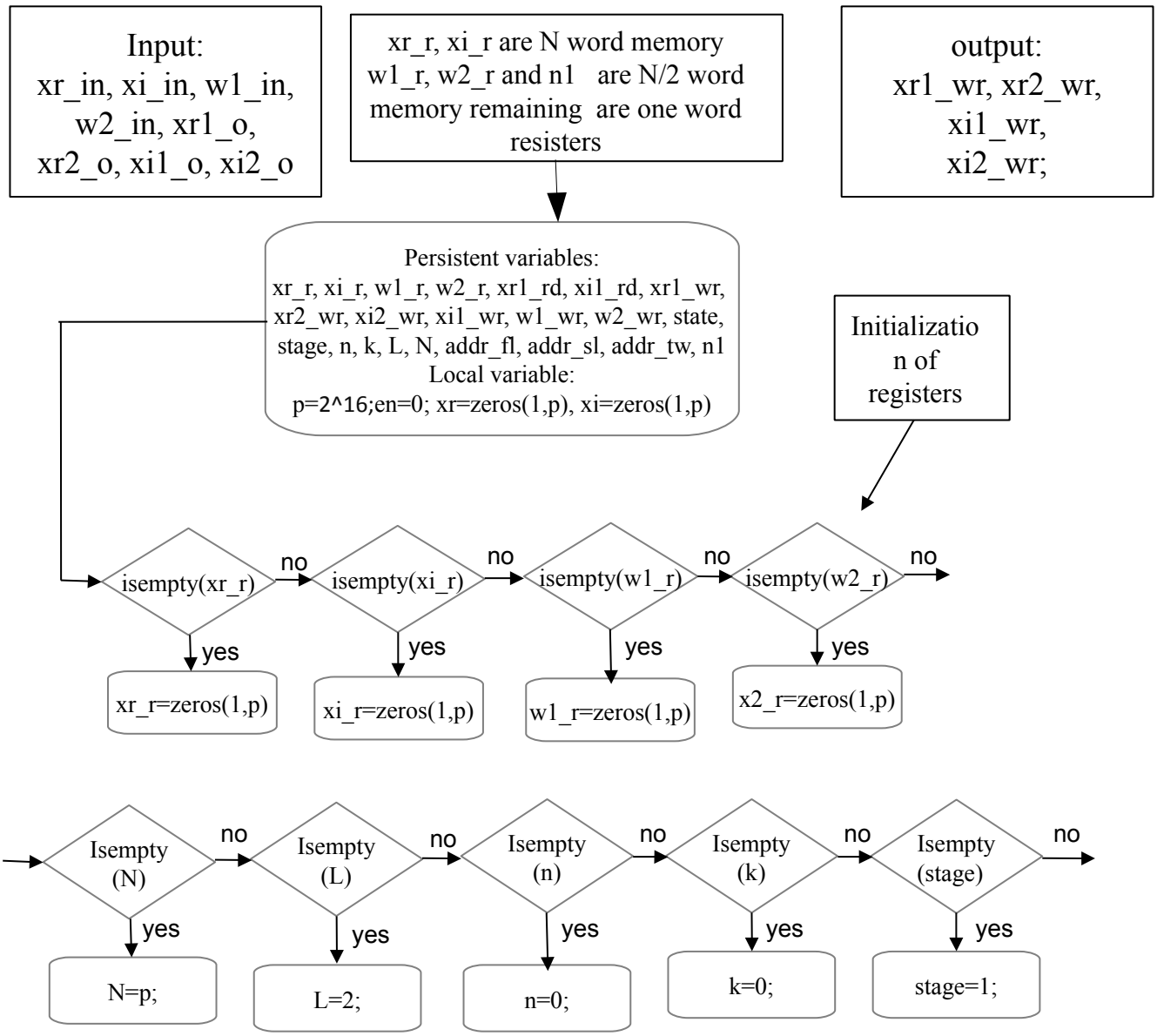

Figure 3.3 FSM Flow chart part-1

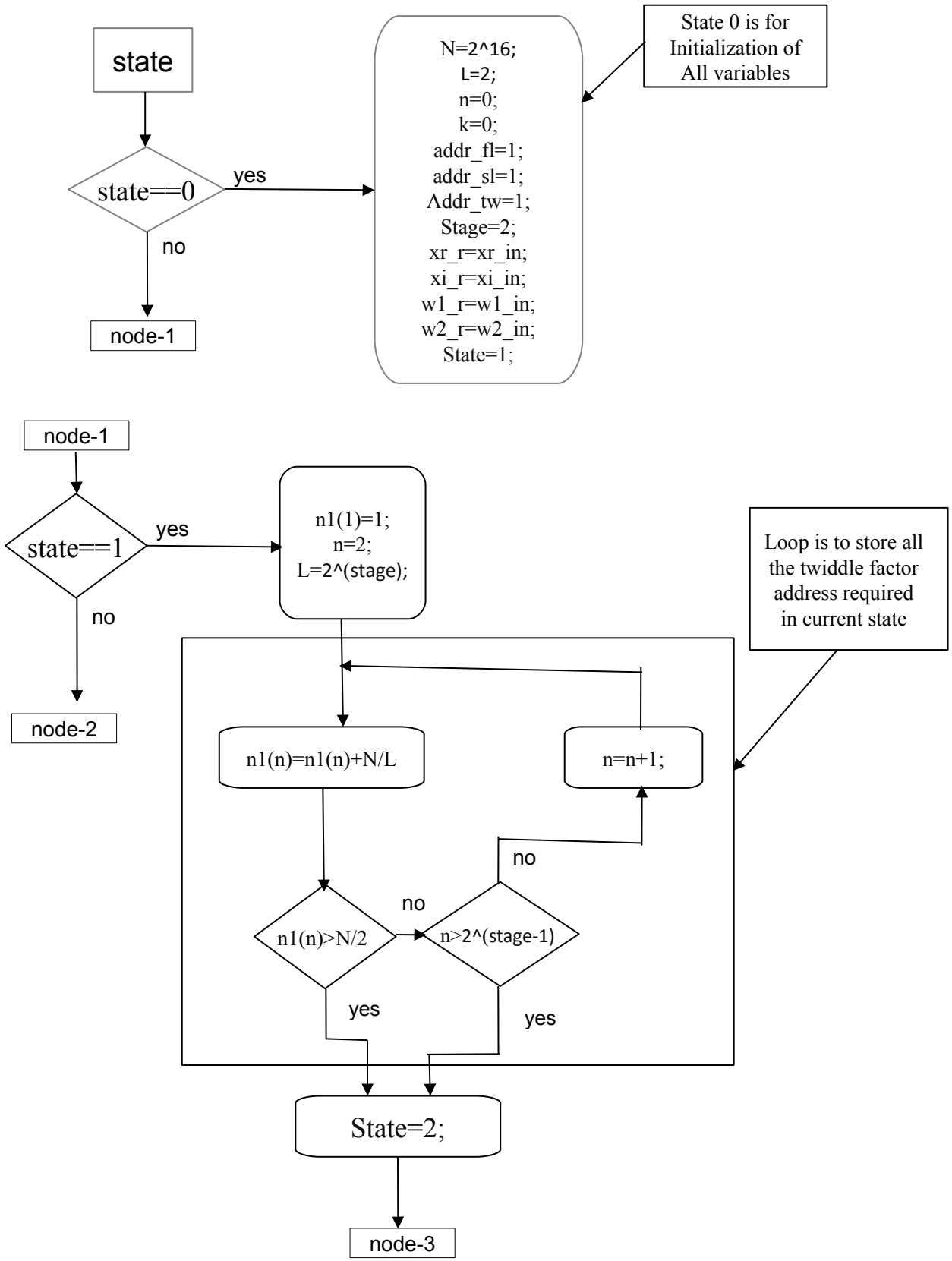

Figure 3.4 FSM Flow chart part-2

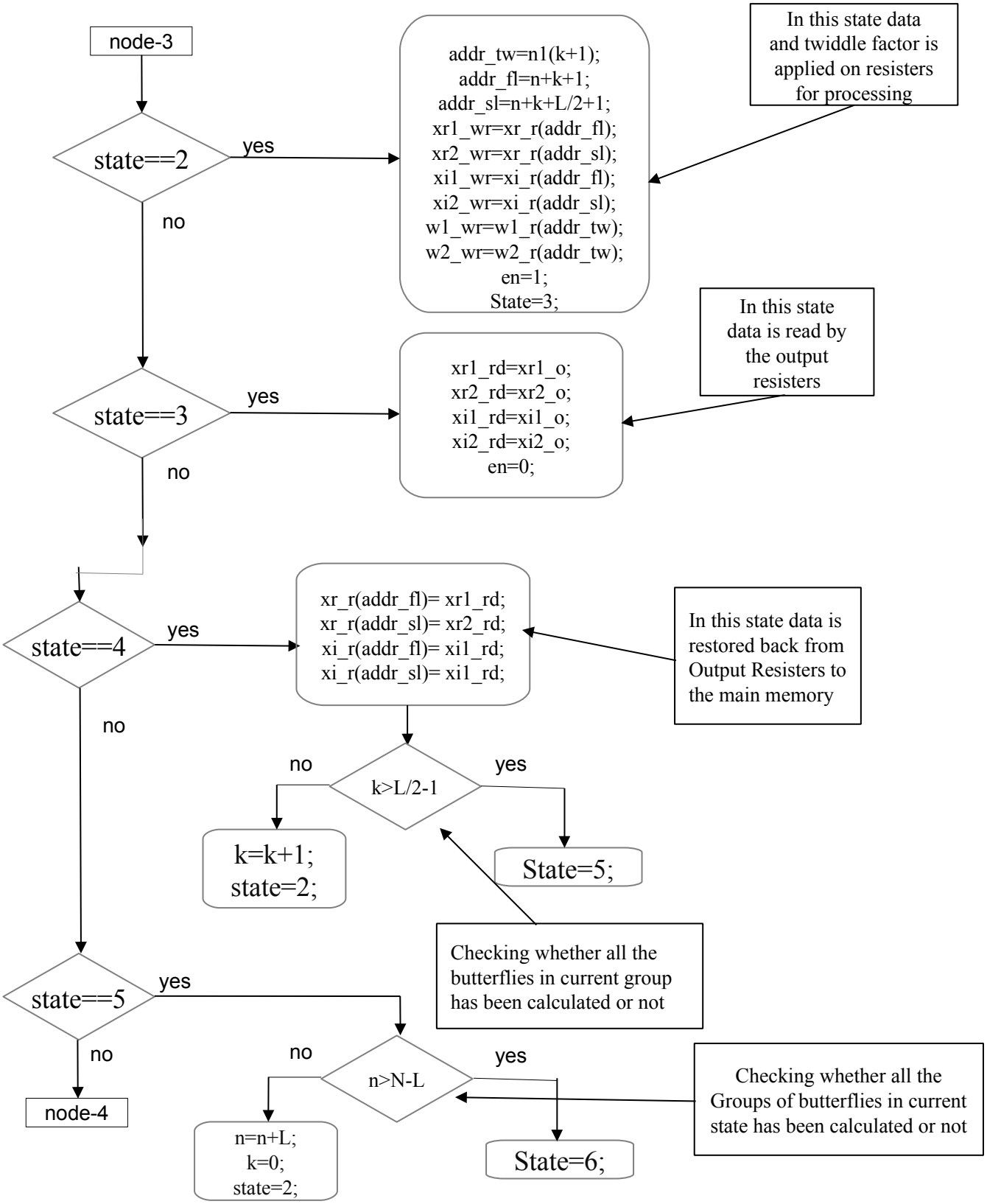

Figure 3.5 FSM Flow chart part-3

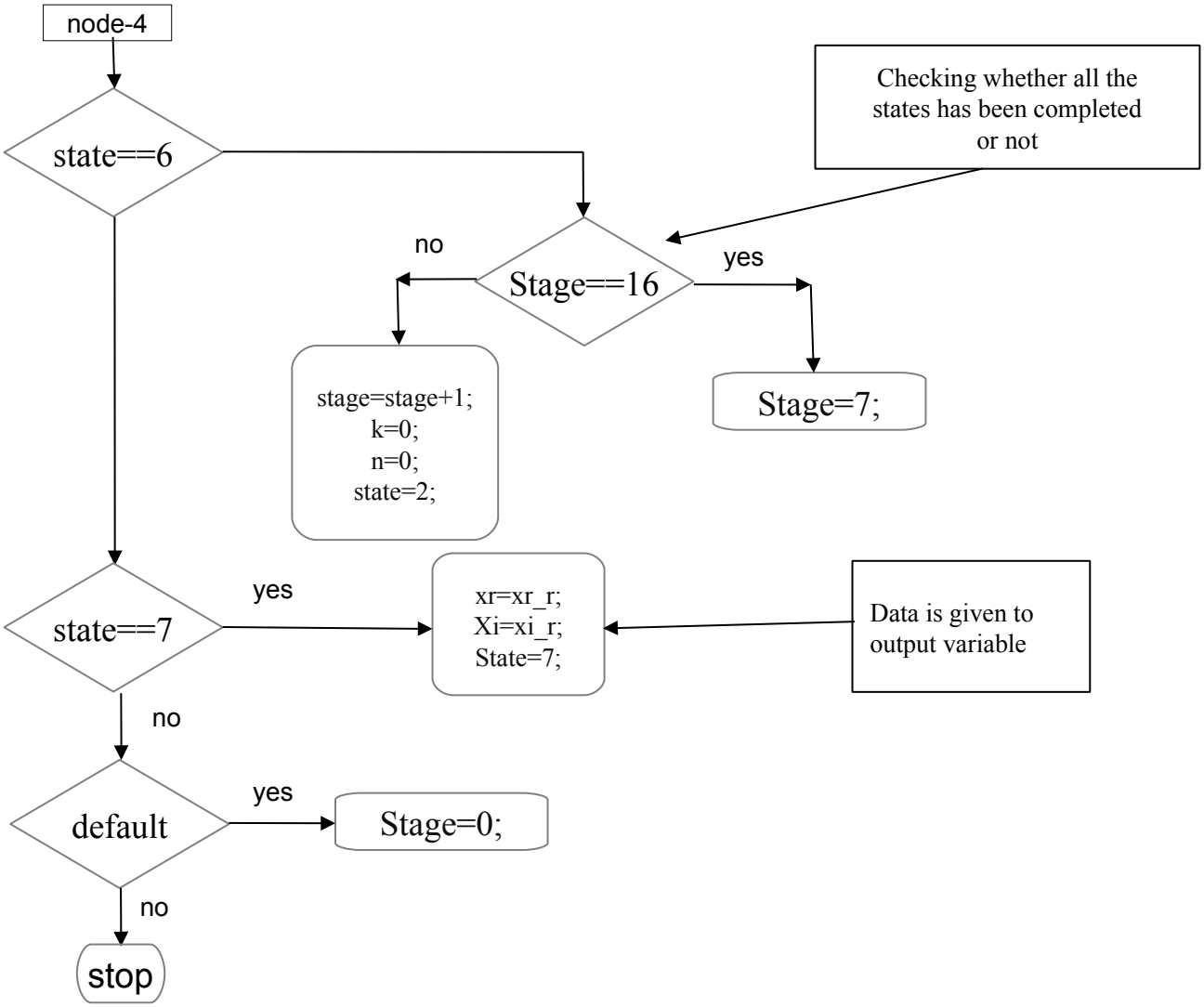

Figure 3.6 FSM Flow chart part-4

3.4.2 Changing the flow chart for radix-4 algorithm

- Now total stages will be N/4
- In each stage total number butterflies will be log4N
- k will vary ask=0:L/4
- n will vary as n=0:L:N-L
- L=4^stage

- tw_1=tw(1:N/L:N/4)
- tw_2=tw(1:2*N/L:N/2)
- tw_3=tw(1:3*N/L:3*N/4)
- addr_fl=n+k+1
- addr_sl=n+k+L/4+1
- addr_tl=n+k+2*L/4+1
- addr_fthl=n+k+3*L/4

3.4.3 Radix-4 butterfly

Memory based architecture requires only one Radix-4 butterfly to compute FFT of the signal. A radix-4 butterfly needs three twiddle factor multiplier to process the data. Twiddle factor multiplication is nothing but the rotation of a vector with given angle. CORDIC algorithm is used to rotate the vector and it avoids the need of complex multipliers and saves huge memory.

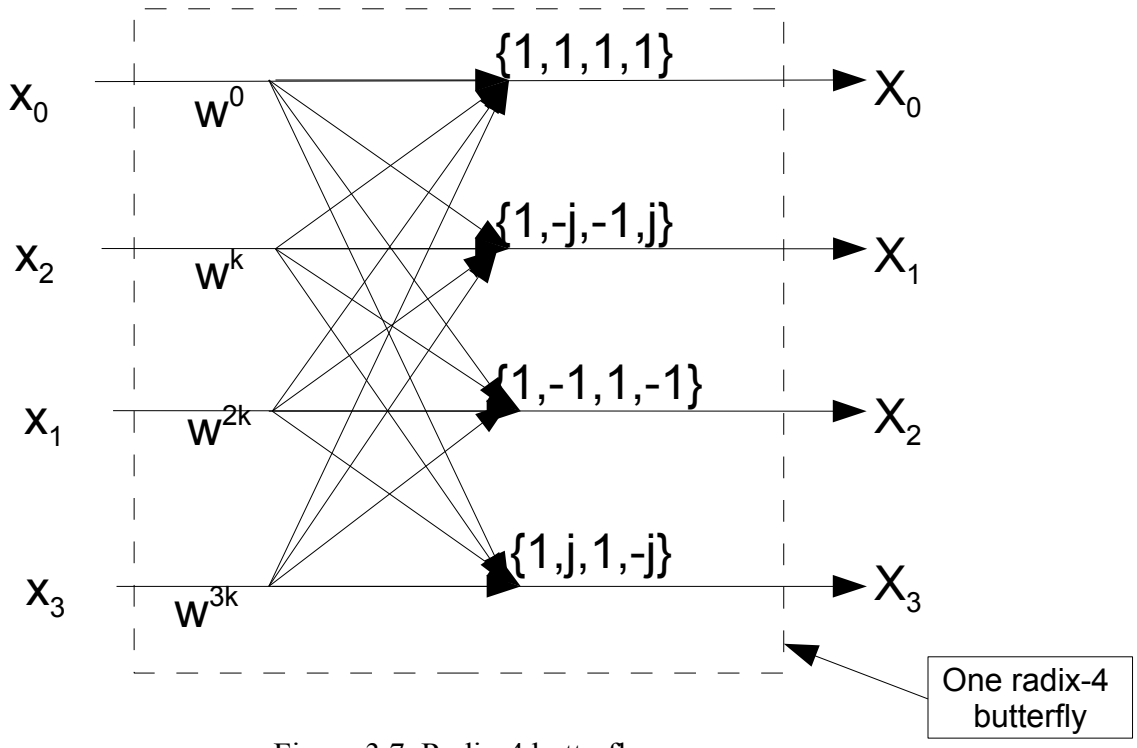

Figure 3.7 Radix-4 butterfly

Next chapter is entirely dedicated to CORDIC algorithm and its implementation. CORDIC algorithm is also used for the implementation of Logarithmic function, Absolute function and square-root function.

Figure 3.7 shows a radix-4 butterfly and Figure 3.8 shows the radix-4 butterfly implemented in Simulink. It requires three CORDIC multipliers, 24 Adders/Subtractors and data shifters. Data shifters divides the data by four to normalize the FFT after computation each butterfly.

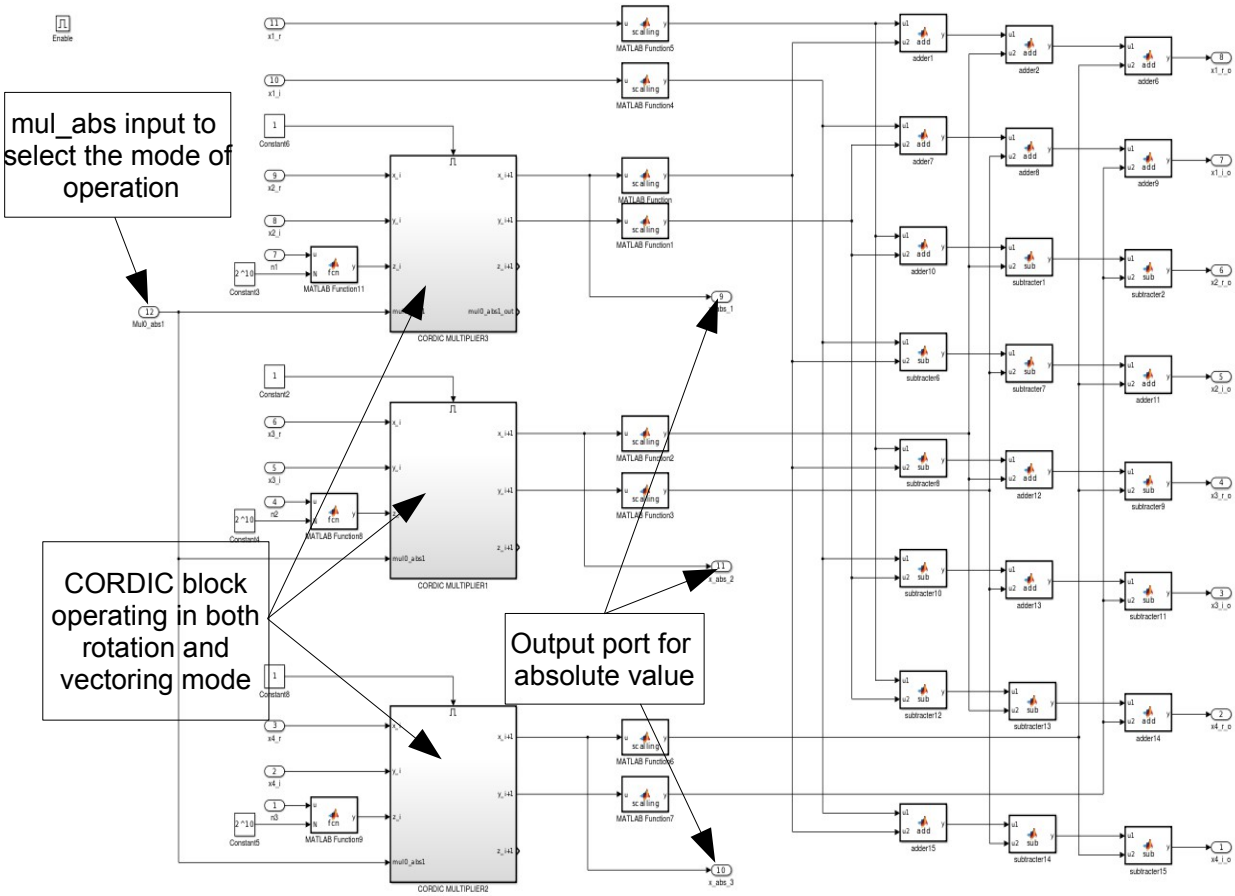

Figure 3.8 Implementation of radix-4 butterfly in Simulink

In Figure 3.7, one can observe that three twiddle factor multipliers are required to multiply the data with W^k, W^{2k} and W^{3k} . Multiplication with $\pm$ 1 and $\pm$ j is nothing but addition and subtraction of

data. Figure 3.8 shows the implementation of radix-4 butterfly using CORDIC multiplier. Same CORDIC multiplier is capable of computing the absolute value of vector (complex number). 'mul_abs' pin selects the desired operation.

There is an angle multipliers corresponding to each CORDIC multiplier. Angle multiplier is converting the address index into angle of rotation by multiplying the address index by 2*pi/N where 'N' is the number of FFT points.

3.4.4 Simulink model of FFT block

Figure 3.9 shows the implementation of FFT Block in Simulink. As discussed earlier, it consist of a radix-4 butterfly with CORDIC multiplier and one control logic block. In this model an array of persistent variable has been used as RAM that stores the input samples before and after processing of data.

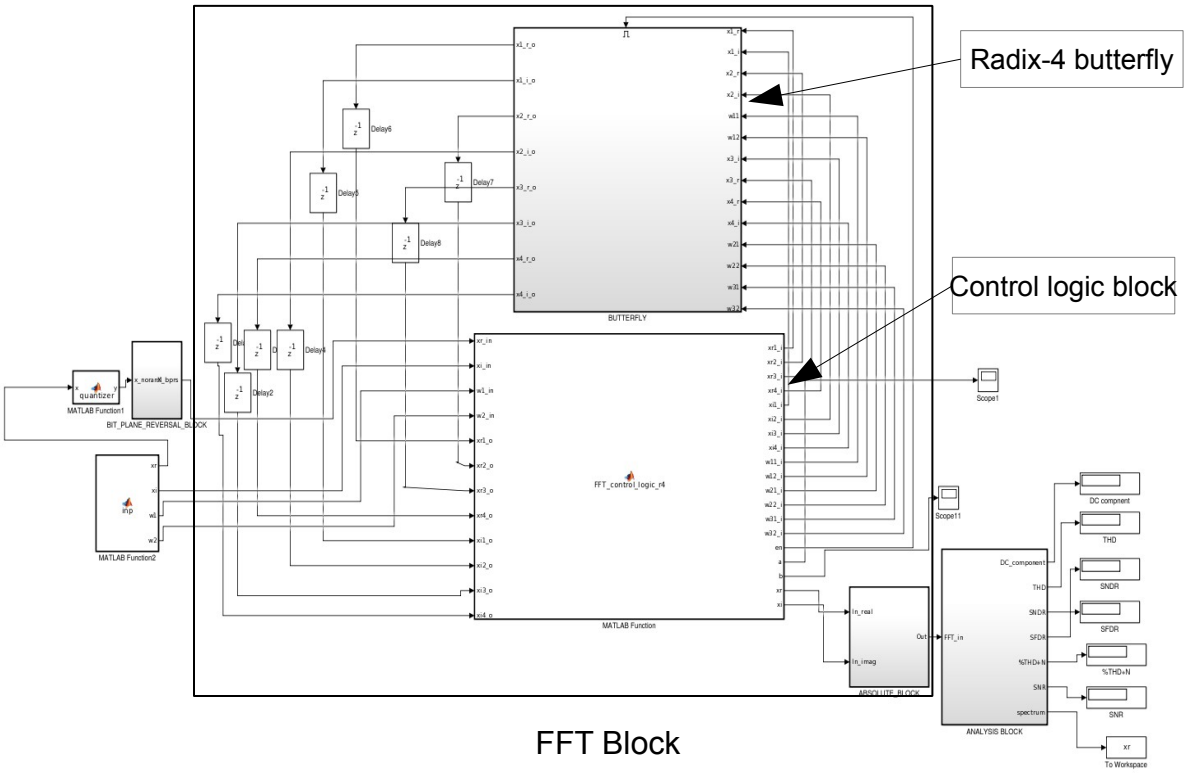

Figure 3.9 FFT block in Simulink

In next level of implementation, a memory block consisting of two RAMs and memory control logic will be inserted and both the FFT block and Analysis block will store and fetch the data from memory block.

3.5 Implementation of Analysis Block in Simulink

Analysis Block consist of a FSM, one adder, one subtractor, one divider, one multiplier, one harmonic generation block and one logarithmic block. FSM reuses these hardware during the analysis. Since in this level of implementation in place of RAM,an array of persistent variable is used but in next level of implementation, Memory block will be inserted at top level consisting of RAMs and analysis block will fetch and store the data from Memory block.

One has to provide the sampling frequency, input signal frequency and number of FFT points as input to the Analysis Block. Using these parameters it performs the analysis.

CORDIC algorithm is used to implement the logarithmic function. This algorithm is discussed in the next chapter.

3.5.1 Flow chart of FSM used in Analysis Block

Figure 3.10 to Figure 3.15 excluding Figure 3.12, show the flow chart of FSM. FSM flow chart is showing the algorithm used for the Analysis Block. Analysis block is computing 'n' number of harmonics in the flow chart but finally the system has been designed for 9 harmonics. Part-1 of the flow chart is showing the computation of all the 9 harmonic frequencies where the algorithm will search the harmonic pattern. Computed harmonic frequencies are divided by the frequency resolution to get the bin corresponding to that frequency. By assuming the peak bin as peak of harmonic, it computes the range of harmonic pattern. Both side of the peak bin it searches the pattern shown in Figure 3.12. It searches the right hand side valley and assumes it as lower range of harmonic and left hand side valley as upper range of harmonics. Then it checks the case of repetition of any frequency. If repetition founds then it sets the flag 'f' to make the computed power zero.

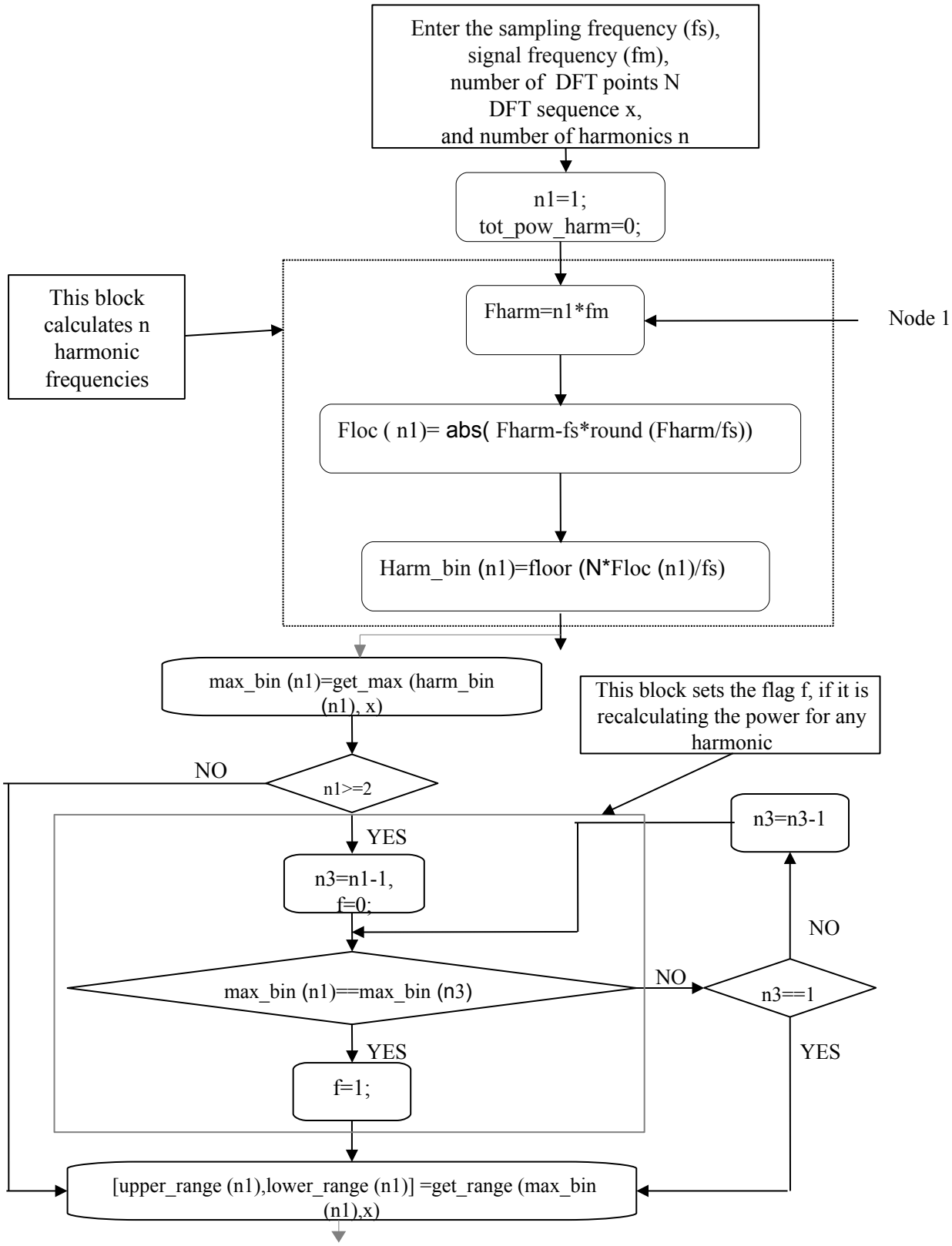

Figure 3.10 FSM Flow chart For the Analysis Block part-1

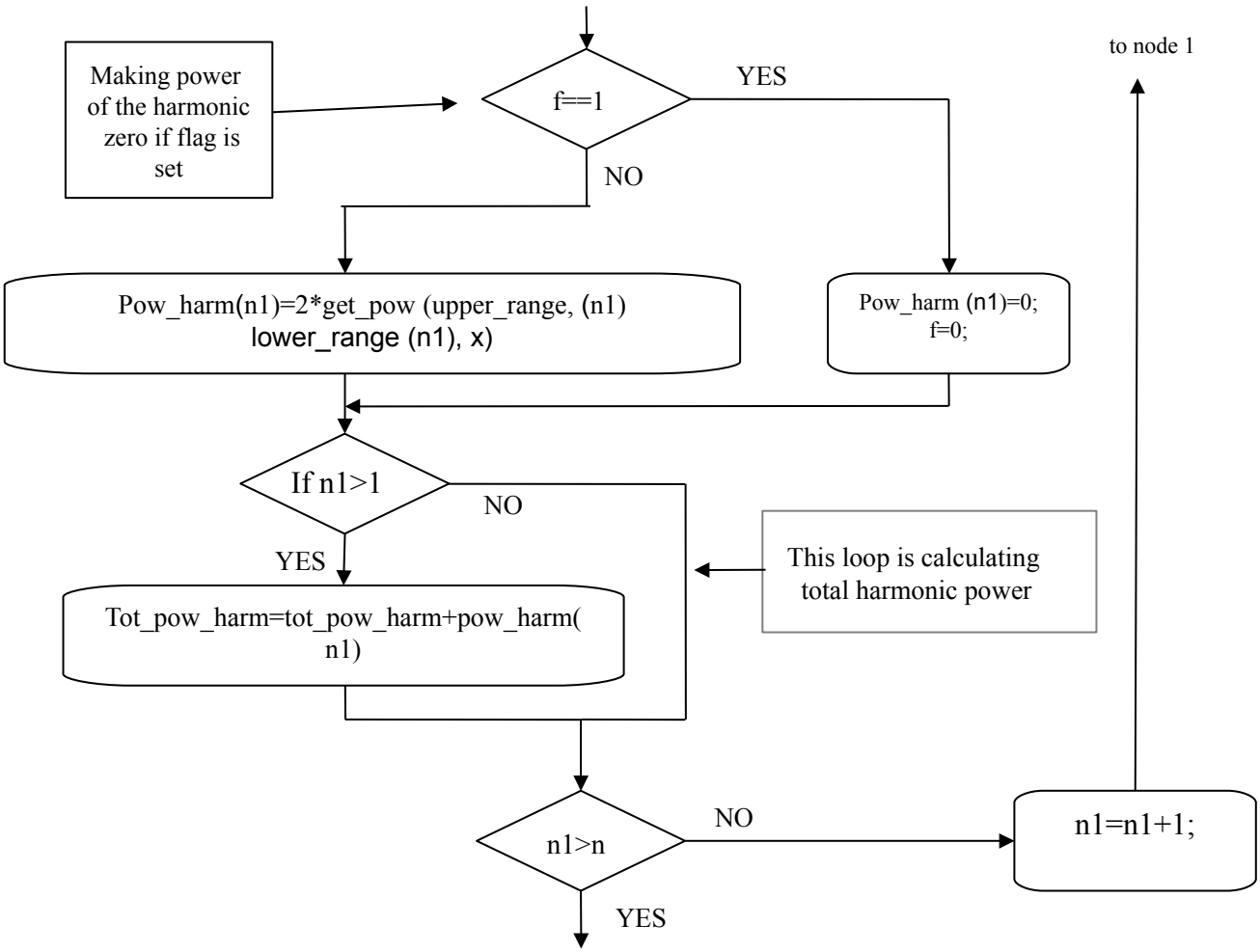

Figure 3.11 FSM Flow chart For the Analysis Block part-2

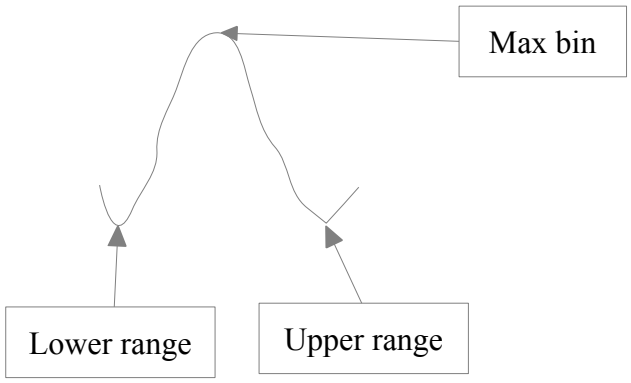

Figure 3.12 Harmonic pattern to compute the range

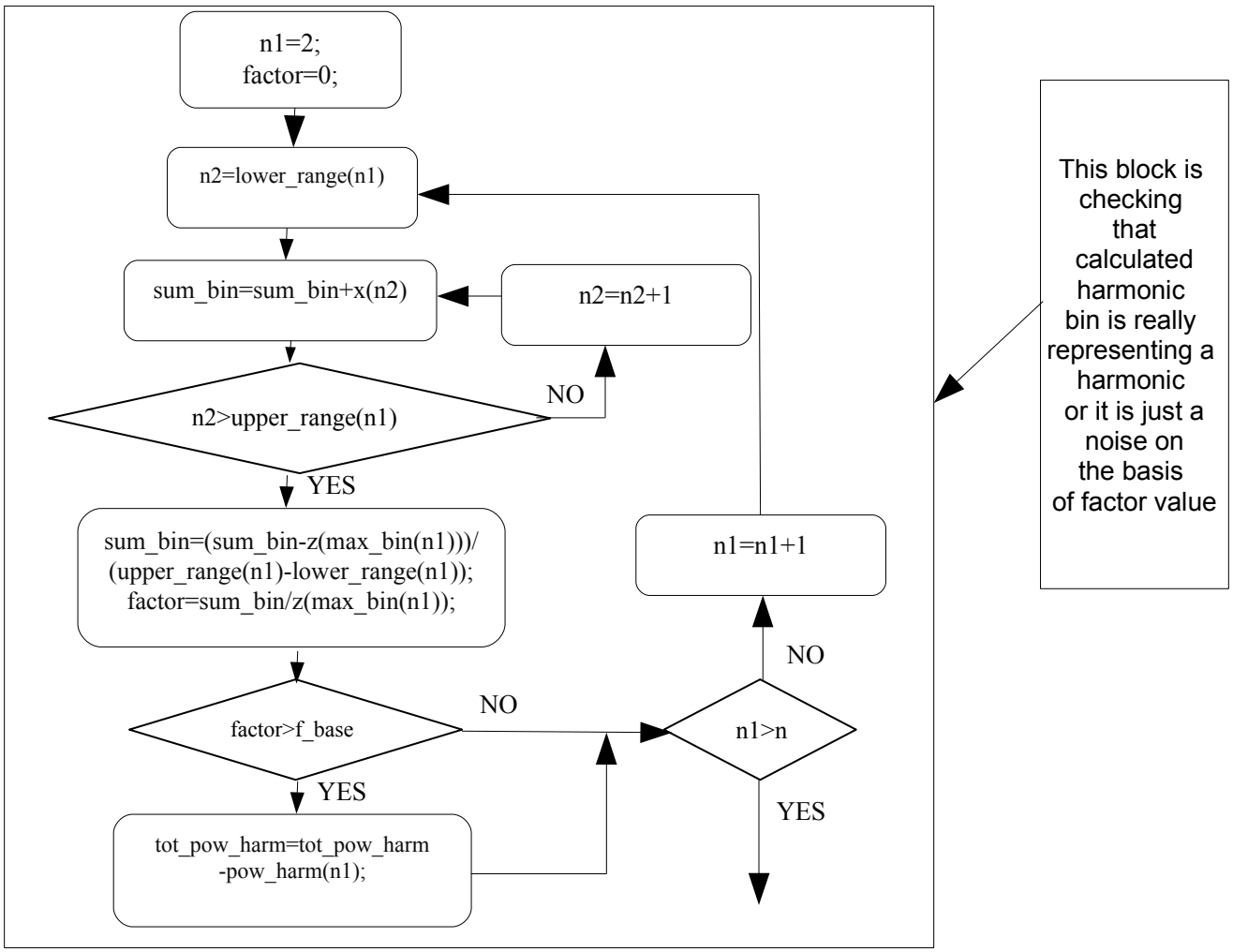

Figure 3.13 FSM Flow chart For the Analysis Block part-3

Part-3 of the flow chart decides the status of computed harmonic on the basis of input 'f_base' and compute factor. If computed factor is greater than 'f_base' the then the computed pattern is assumed to be harmonic otherwise it is assumed as noise and its power does not add into total harmonic power. Part-4 of the algorithm is computing the total power and subtracting the signal power from computed total power and calculates total noise plus distortion power. In the next step it computes SNDR, THD, and SNDR. To calculate the SFDR, it searches the maximum noise amplitude at right side of signal lower range. In part-4, it searches the left hand side of signal upper range and compute the SFDR.

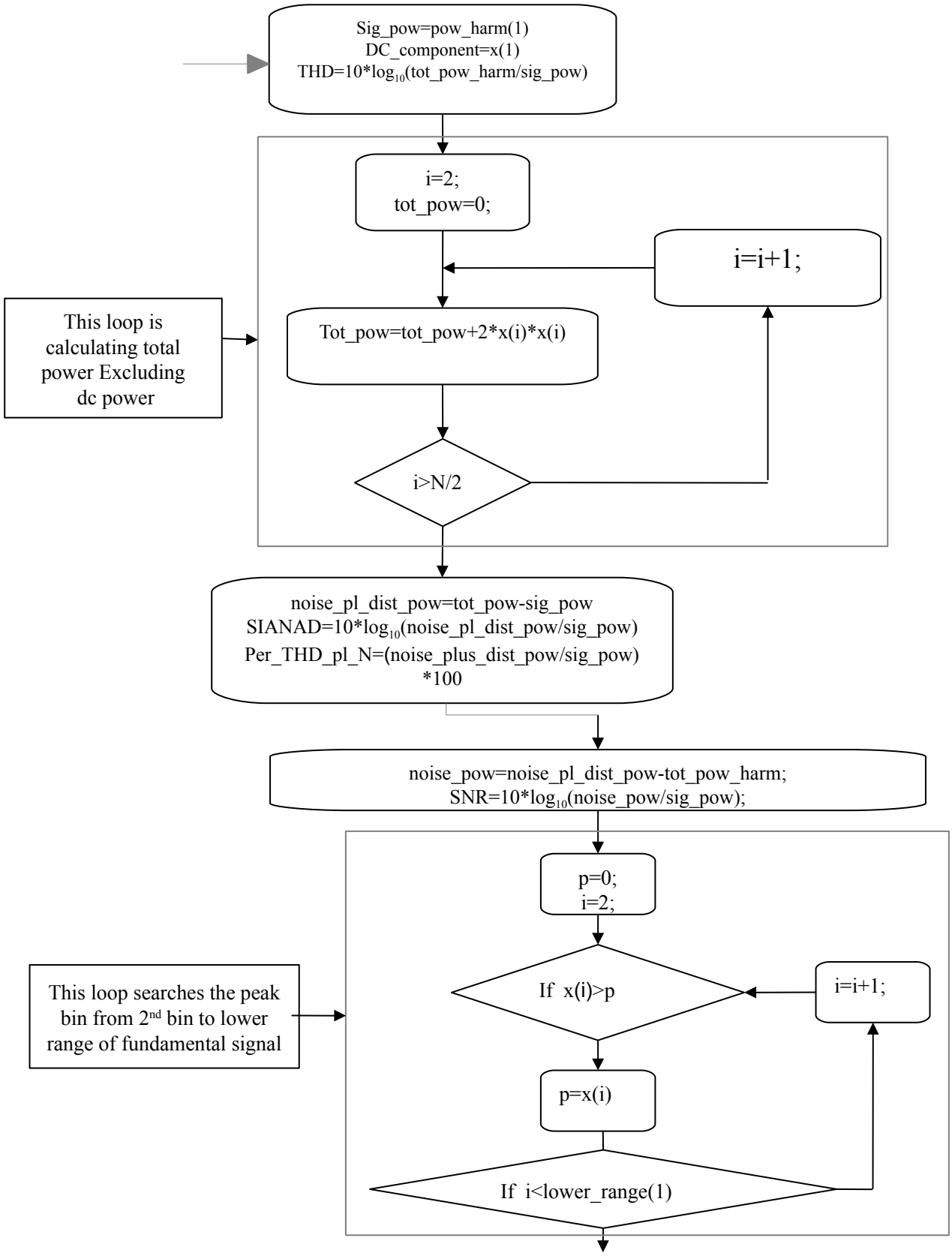

Figure 3.14 FSM Flow chart For the Analysis Block part-4

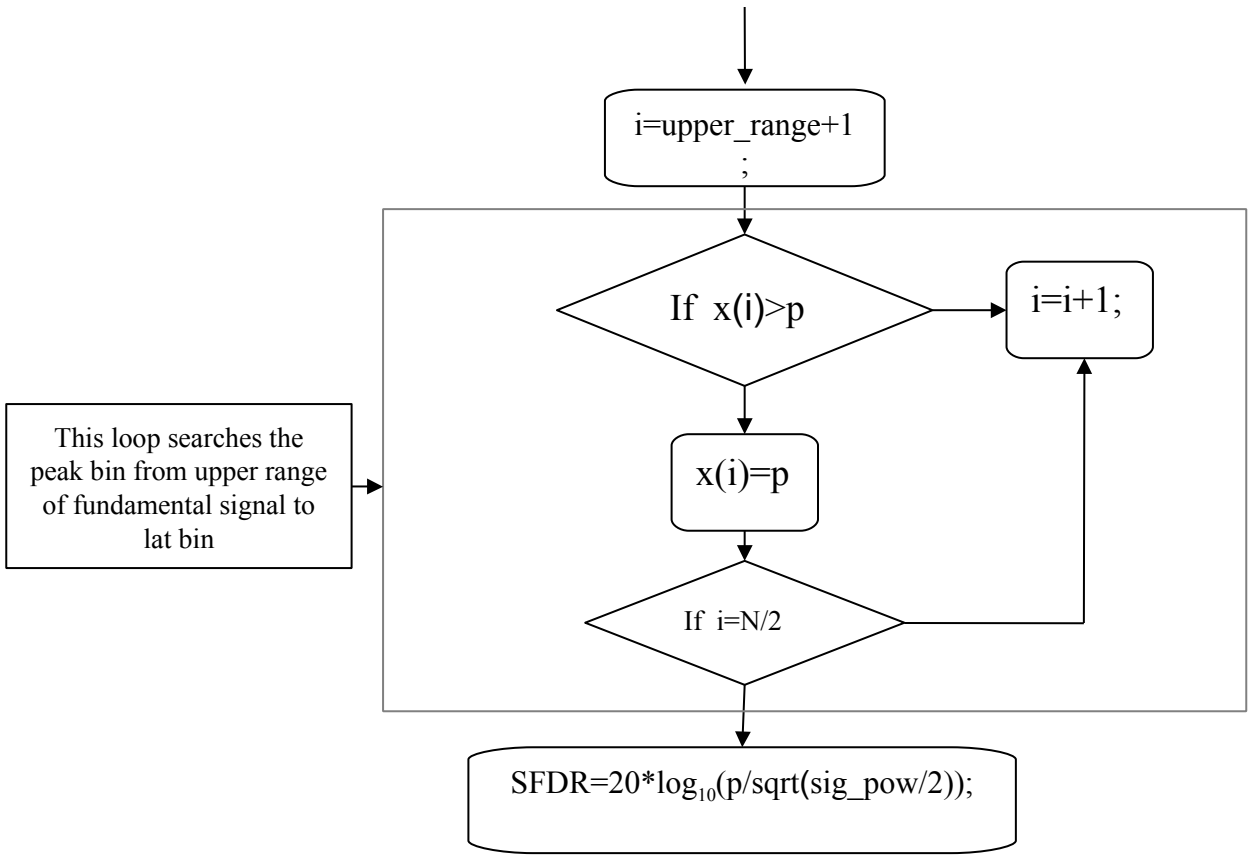

Figure 3.15 FSM Flow chart For the Analysis Block part-5

3.5.2 Other resources used in Analysis Block

Apart from FSM, Analysis Block consist of Adder, Subtractor, Multiplier, Divider, Harmonic bin generation block and logarithmic block to perform the analysis. Logarithmic block will be discussed in next chapter. Harmonic bin generation block takes the input signal frequency and index of harmonic and calculates the expected peak bin of harmonic. Harmonic frequencies that goes beyond the first Nyquist-zone, it folds back those frequencies to first Nyquist-zone using the formula given by Equation (3.1).

$$Fharm=abs(Fin*i - Fs*abs(Fin*i/Fs)) \quad (3.1)$$

Fharm is the Harmonic frequency in first nyquist-zone, Fin is input signal frequency, I is index of harmonics and Fs is sampling frequency.

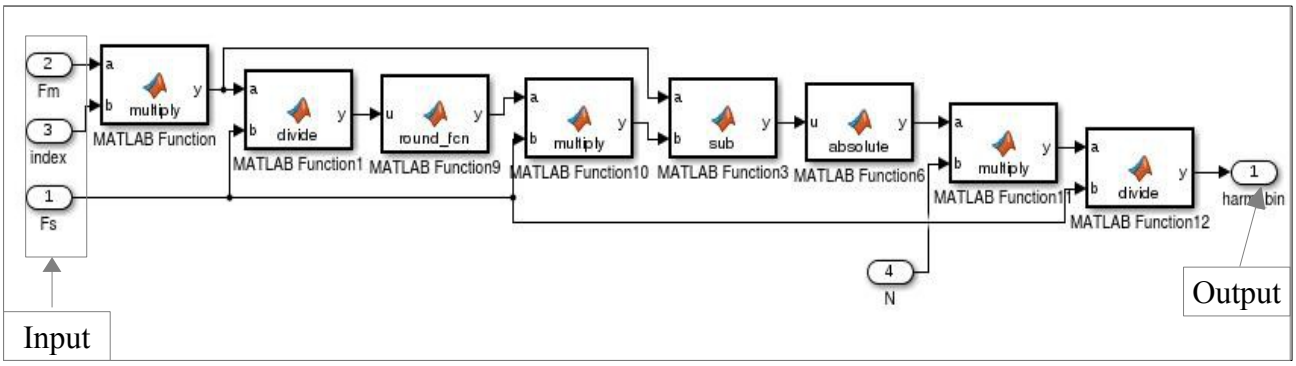

Figure 3.16 Harmonic bin generation block

Figure 3.16 shows the harmonic bin generation block. It also consist of adders multiplier, subtractors and absolute block. In next step of implementation, harmonic bin generation block will be removed and harmonic bin will be generated using the available adders and subtractors.

3.5.3 Simulink model of Analysis block

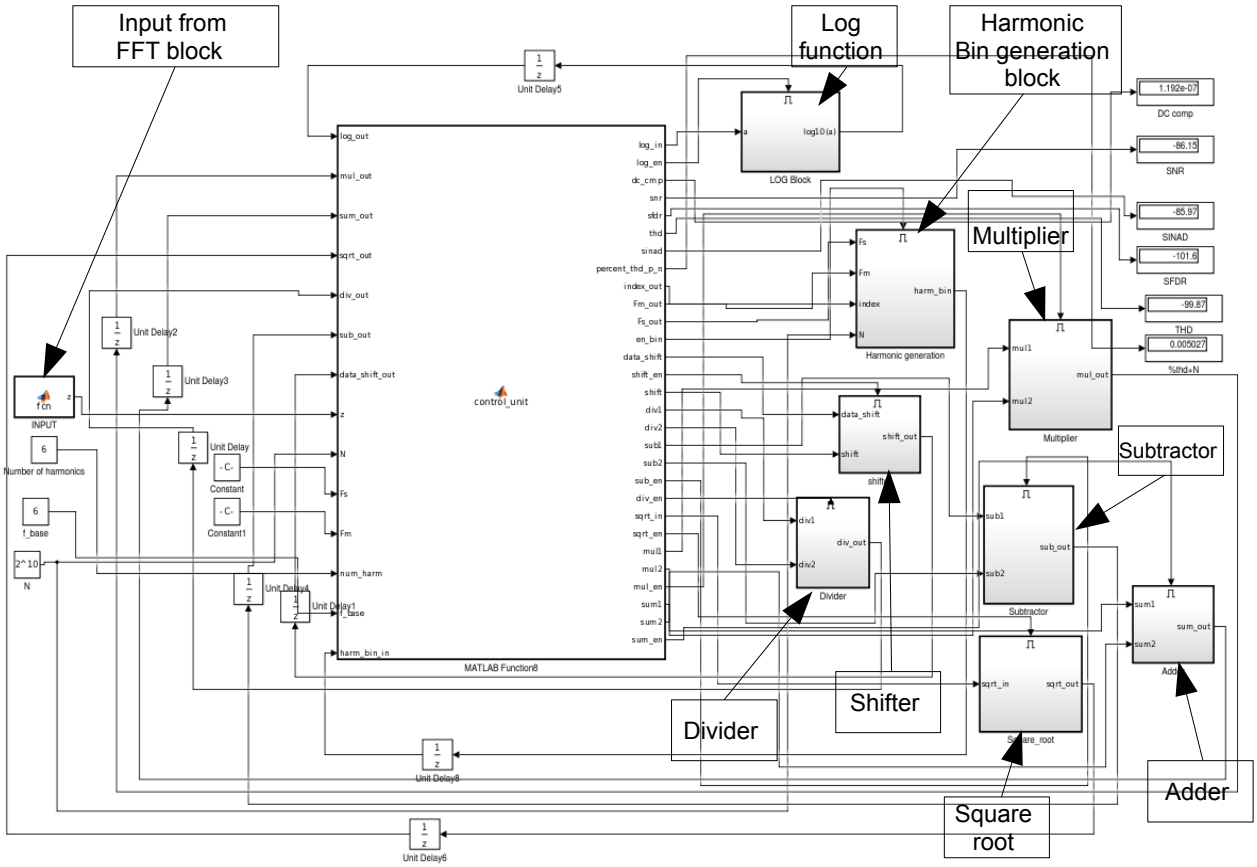

Figure 3.17 Analysis Block in Simulink

Figure 3.17 shows the analysis block designed in Simulink. Simulink model shows that analysis block needs following hardware:

1. Control logic bloc
2. Harmonic bin generation block
3. Logarithmic block
4. Addder, Subtracter, Multiplier, Divider, Shifter

Chapter-4

CORDIC Algorithm

4.1 Objective

There are three main objectives of CORDIC Algorithm in the project work:

1. Our first objective is to make a complex multiplier (or to rotate a given vector having coordinate (x, y) with any arbitrary angle θ) using CORDIC algorithm.
2. Our second objective is to convert a given vector having coordinate (x, y) from Cartesian form to polar form (or to get the absolute value of a given vector and angle made with positive x axis).
3. Our third objective is to use the CORDIC algorithm for the implementation of logarithmic and square-root function.

We can make a complex multiplier with real adders and subtractors but in FFT implementation we just need to rotate a complex number with a given angle so we are just changing the phase of that complex number and magnitude of that number is always constant. If x(n) is some input sequence which length is N and X(k) is its DFT then we can use Equation 4.1 to find out X(k) as

$$X(k)=\sum_{n=0}^{n=N-1} x(n)e^{(-j\frac{2\pi}{N}(n.k))} \tag{4.1}$$

Here x(n) is a complex variable and we are changing its phase for every variation of n and k. Since magnitude of the product is same but the magnitude of its real and imaginary component is changing because of rotation.

Figure 4.1 is showing the radix-2 butterfly for decimation in time FFT algorithm. x1and x2 are two input data and X1 and X2 are its 2 point FFT.

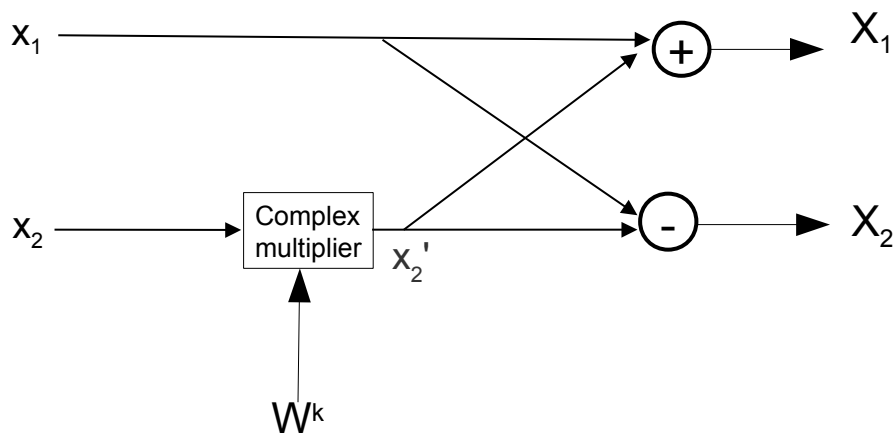

Figure 4. 1 radix-2 butterfly (decimation in time)

Where $W^k = e^{-2*pi*k / N}$ and it is called twiddle factor.

Now we have two options

1. W e can find out equivalent real and imaginary part corresponding to W^k as given in Equation 4.2.

$$Wk=\cos(2*pi*k / N) - j\sin(2*pi*k / N) \tag{4.2}$$

And assume x1 and x2 are two complex number as given in Equation 4.3 and Equation 4.4.

$$x1=x_1r+jx_1i \tag{4.3}$$

And

$$x_2=x_2r+jx_2i \tag{4.4}$$

Now after complex multiplication x'_2 will also be complex number as shown in Equation 4.5.

$$x'_2=x'_2r+jx'_2i \tag{4.5}$$

Then real and imaginary part of x'_2 will be given by Equation 4.6 and Equation 4.7

$$x'_2r = x_2r * \cos(2*pi*k / N) + x_2i * \sin(2*pi*k / N) \quad (4.6)$$

And

$$x'_2i = x_2i * \cos(2*pi*k / N) - x_2r * \sin(2*pi*k / N) \quad (4.7)$$

We can store the values of sin(2*pi*k / N) and cos(2*pi*k / N) for each value of k and in this case we can implement real and imaginary part using real adders and subtractors and multipliers.

2. We can use CORDIC multiplier where in place of applying real and imaginary part of W^k we have to apply the angle through which vector is rotating that is 2*pi*k / N. So there is no need to store the real and imaginary values of twiddle factor. We can also use this algorithm to compute absolute value of a vector (complex number).

4.2 Introduction to CORDIC algorithm

CORDIC algorithm can compute trigonometric and transcendental function using only basic hardware as adders, shifters and multiplexers . The CORDIC algorithm is used for the computations in most hand-held calculators for the computation of transcendental functions. It can also be used for the computation of twiddle factor and implementation of logarithmic function. Basically

1. It can compute trigonometric functions as cos, sin, arctan
2. It can compute hyperbolic trigonometric functions as, cosh, sinh, arctanh
3. It can compute Logarithmic functions (ln, log) and square-root function
4. It can perform complex multiplication and can also compute the absolute value of a vector

4.3 Concept of CORDIC Algorithm

Basic idea of CORDIC algorithm is to rotate a vector with an arbitrary angle by rotating the

vector with some previously fixed angles and choosing the those fixed angles in such a way that can be easily realized using simple hardware.

Suppose we want to rotate a vector with co-ordinate (x, y), by an arbitrary angle θ then Equation 4.8 is showing coordinates of resultant vector (x', y')

$$x' + jy' = (x+jy)e^{j\theta} = (x+jy)(\cos\theta + j\sin\theta) \tag{4.8}$$

And Equation 4.8 and Equation 4.9 and Equation 4.10 is showing the real and imaginary part of vector separately.

$$x' = x.\cos\theta - y.\sin\theta \tag{4.9}$$

$$y' = x.\sin\theta + y.\cos\theta \tag{4.10}$$

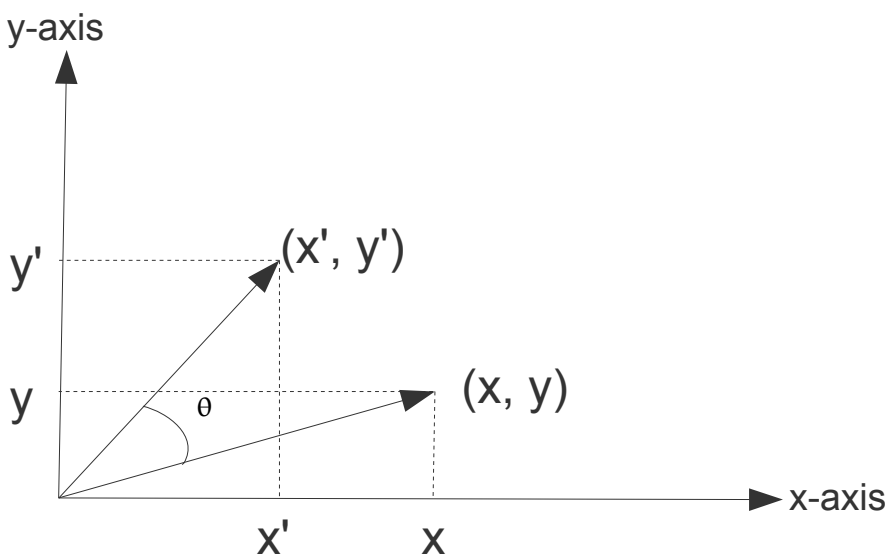

Figure 4.2. Rotation of a vector having coordinate (x, y) with a fixed angles θ

So if we know the sine and cosine value corresponding to the arbitrary angle θ then we can find out the next vector after rotation. But storing sine and cosine values require larger memory. So we have to simplify this equation.

Equation 4.9 and Equation 4.10 can be rearranged can be given by Equation 4.11 and 12.

$$x' = \cos\theta\,(x - y.\tan\theta) \tag{4.11}$$

$$y' = \cos\theta\,(y + x.\tan\theta) \tag{4.12}$$

In Figure 4.2, we can see that angle corresponding to vector (x' , y') will be given by Equation 4.13

$$\theta' = \tan^{-1}\frac{(x - y.tan\theta)}{(y + x.tan\theta)} \tag{4.13}$$

So we can see that tanθ is responsible for the change in angle of next rotated vector and cosθ is just changing the magnitude of next vector.

Now if we store the values of tanθ in each iteration according to the series $\pm 45*2^{-i}$, then we can rotate the vector by an arbitrary angle θ by rotating the vector with fixed angle in each step. But for this we need two multipliers at each stage to for each x' and y' to get new value after iteration.

We can show one example showing that any angle between -90 to +90 can be realized using series that takes a step of $\pm 45*2^{-i}$ where i is representing the ith step of series. Whether we will go for a positive sign or we will go for negative sign this will be decided by the difference of target angle and sum of current angles (or current position of approximation) . In order to trace this difference we can define a variable z and we can also define a variable d that will take a value (either +1 or -1) based on the sign of variable z.

So we can trace the variable z for each step until we reach within error limit. In the Table 4.1 our target angle is 30 degree and a(i) is representing the step in i^{th} iteration and d(i) is deciding that whether step will be in clockwise direction (negative) or anti-clockwise direction (positive).

i	d(i)	z(i)	a(i)
0		30	45
1	1	-15	22.5
2	-1	7.5	11.25
3	1	-3.75	5.63
4	-1	1.88	2.81
5	1	-0.93	1.41
6	-1	0.48	0.70
7	1	-0.22	0.35
8	-1	0.13	0.18
9	1	-0.05	0.09

Table 4.1 Rotating by 30 degree using fixed angles according to series

For the next stage, z(i+1) and d(i+1) are given by Equation 4.14 and Equation 4.15

$$z(i+1)=z(i)-\ 45*2^{-i} \tag{4.14}$$

and

$$d(i+1)=\begin{cases} +1 & z(i)\geq 0 \\ -1 & z(i)<0 \end{cases} \tag{4.15}$$

In Table 4.1 we can see that in 11 step angle difference is zero or in other words our rotating vector and target vector are same. As we have already seen that if we want to rotate a vector (x,y) by an angle θ then next vector x' and y' can be given by Equation 4.11 and Equation 4.12.

Equation 4.14 is showing the angle z(i) in degree. In Table 4.2 we can see that values of tan-1(2-i)*180/pi and $45*2^{-i}$ are almost on same pattern and since we are tracing the angle difference to get the final vector so difference in both the series will be taken care by variable z. We can take the variable z in radian in place of taking it in degree then our series will be given by Equation 4.16. Advantage of taking the series written in Equation 4.16 is that now tan θ_i =+ 2-i as shown in Equation

4.17. Now multiplication with these values of $\tan \theta_i$ is nothing but shifting of data.

i	45*2^-i	$\tan^{-1}(2^{-i})$*180/pi	Tan^{-1}(2^-i)	2^-i	Error
0	45	45	7.85E-001	1.00E+000	-2.15E-001
1	22.5	26.57	4.64E-001	5.00E-001	-3.64E-002
2	11.25	14.04	2.45E-001	2.50E-001	-5.02E-003
3	5.63	7.13	1.24E-001	1.25E-001	-6.45E-004
4	2.81	3.58	6.24E-002	6.25E-002	-8.12E-005
5	1.41	1.79	3.12E-002	3.13E-002	-1.02E-005
6	0.70	0.90	1.56E-002	1.56E-002	-1.27E-006
7	0.35	0.45	7.81E-003	7.81E-003	-1.59E-007
8	0.18	0.22	3.91E-003	3.91E-003	-1.99E-008
9	0.09	0.11	1.95E-003	1.95E-003	-2.48E-009
10	0.04	0.06	9.77E-004	9.77E-004	-3.10E-010
11	0.02	0.03	4.88E-004	4.88E-004	-3.88E-011
12	0.01	0.01	2.44E-004	2.44E-004	-4.85E-012
13	0.01	0.01	1.22E-004	1.22E-004	-6.06E-013

Table 4.2 Fixed angles for both the two series and the error between 2^{-i} and $\tan^{-1}(2^{-i})$

So CORDIC algorithm can replace the need of multipliers into data shifters by taking the incremental angle at each stage as

$$\theta i = \pm \tan^{-1} 2^{-i} \tag{4.16}$$

$$\tan \theta_i = \pm 2^{-i} \tag{4.17}$$

Now arbitrary angle is given by Equation 4.18 and the difference of angle at each stage z(i) will be given by Equation 4.19 as

$$\theta = \sum_{i=0}^{\infty} \theta_i \tag{4.18}$$

where

$$z(i+1) = z(i) - \theta_i \tag{4.19}$$

So if we will go for infinite no of rotation then the difference will be definitely zero but for finite number of stages, at each stage angle difference will be given by Equation 4.20.

$$z(i+1) - z(i) = -\theta_i = -d_i * \tan^{-1}2^{-i} \qquad (4.20)$$

So after i^{th} iteration error will be $\tan^{-1}2^{-i}$ and we can see in Table 4.2 that for i>10

$$\tan^{-1}2^{-i} \approx 2^{-i} \qquad \text{(with an error, less than } 2^{-10}\text{)}$$

After i^{th} iteration error will be approximately 2^{-i} (here we are neglecting the error due to magnitude compensation since after 10-iterations error due to magnitude compensation will be approximately of the order 2^{-20} explain in Table 4.3).

The fixed angles can be stored in a LUT (Look Up Table) and directly applied to adders / subtractors to get the next value of z. we can calculate number of stages (or iterations) required to make the error within an error limit.

Now in Equation 4.11 and Equation 4.12, if we replace the value of $\tan\theta_i$ as given in equation -17 then new equations will be given by Equation 4.21 and 4.22.

$$x(i+1)=\cos\theta_i\,[x(i) - d_i.\ y(i).2^{-i}] \qquad (4.21)$$

$$y(i+1)=\cos\theta_i\,[y(i) + d_i.\ x(i).2^{-i}] \qquad (24.2)$$

And Equation 4.20 will remain same and next value of variable d will be given by either Equation 4.28 or Equation 4.29, depending on the mode of operation(it will be discussed in next section).

Here we can observe that there is no need of multipliers since multiplication with 2^{-i} is nothing but shifting the word by i bits.

One final observation is that since in each iteration we are multiplying our data with $\cos\theta_i$ which is just changing the magnitude of the vector, in each iteration if we do not multiply with that then the vector magnitude will increase and multiplication also need a multiplier in each stage.

So we can further simplify the Equation 4.21 and Equation 4.22 by removing this $\cos\theta_i$ terms from the equations and we can calculate the overall magnitude compensation term according to Equation 4.23 as and using Equation 4.24 and Equation 4.25 we can get the value of K.

$$K=\cos\theta_0 * \cos\theta_1 * \cos\theta_2 * \cos\theta_3 * \ldots * \cos\theta_\infty \tag{4.23}$$

and

$$cos\theta_i = \frac{1}{(1+\tan^2\theta_i)^{\frac{1}{2}}} \tag{4.24}$$

so for $\tan\theta_i = \pm 2^{-i}$

$$cos\theta_i = \frac{1}{(1+2^{-2i})^{\frac{1}{2}}} \tag{4.25}$$

K= 0.607252935009...

Finally for magnitude compensation we can multiply the result by K that will compensate the error due to avoiding the multiplications. So our final vector will be given by Equation 4.26 and 4.27.

$$x_{final}=x(i_{final})*K \tag{4.26}$$

$$y_{final}=y(i_{final})*K \tag{4.27}$$

where (x_{final}, y_{final}) is Cartesian form of vector after last rotation.

For magnitude compensation we can also calculate K for finite number of iterations but for

i>10, as we can see from Table 4.3 that error is less than 2^{-23} (error has been calculated using logarithmic with base-2 so that we can see the error in form of 2^{k}), even for 10 stages it needs at least 23-bits to differentiate between k_{ideal} and k_{10} and error is less than 2^{-31} for 14 stages.

Here K_{ideal} is the value of K for infinite iterations (computed using 1000 rotations only). K_i is value of K by considering up to I iterations. So neglecting this error, overall error in new ordinates of rotated vector will be approximately 2^{-i}, after completing i number of iterations.

i	K_i-k_{ideal}	log2(K_i-K_{ideal})
0	9.99E-002	-3.32
1	2.52E-002	-5.31
2	6.32E-003	-7.31
3	1.58E-003	-9.30
4	3.95E-004	-11.30
5	9.88E-005	-13.30
6	2.47E-005	-15.30
7	6.18E-006	-17.30
8	1.54E-006	-19.30
9	3.88E-007	-21.30
10	9.77E-008	-23.29
11	2.41E-008	-25.31
12	6.03E-009	-27.31
13	1.50E-009	-29.32
14	3.77E-010	-31.31

Table 4.3. Error in K by considering infinite iterations (K_{ideal}) and i iterations (K_i)

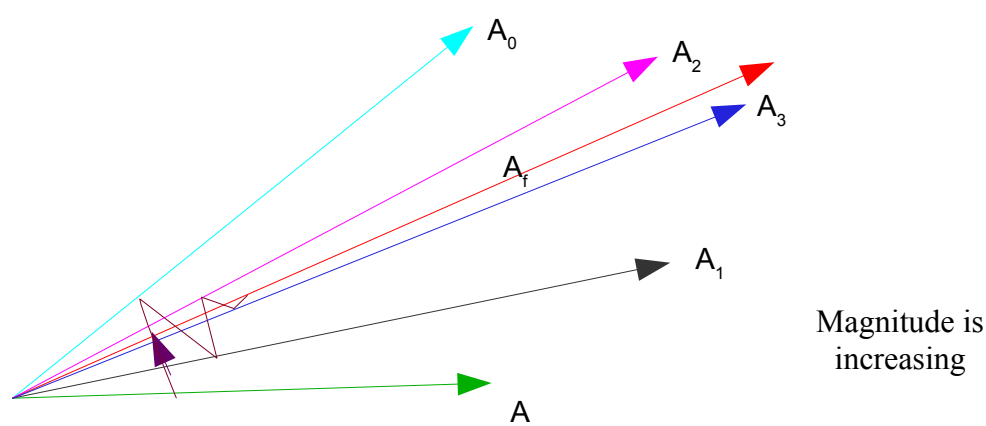

Figure 4.3. Rotation of vector A and A_f is the final vector vector after 4 rotation

We can see in the Figure 4.3, A is initial vector and in i^{th} iteration it has a angle θ_i and it became A_i. A_f (in red) is the final vector. We can see that difference of angle is reducing but due to magnitude error its final amplitude is A/K and after multiplying it by K the amplitude will reduce to A with an error corresponding to the final iteration.

4.4 Modes of operation in circular co-ordinate system

CORDIC algorithm operates in two different modes of operation known as 'Vectoring mode' and 'Rotation mode'. In Rotation mode of operation, algorithm rotates the vector by its initial angle to compute the magnitude of input vector and algebraic sum of all the fixed angle gives the initial angle of input vector. In the vectoring mode of operation, user provides the co-ordinate of input vector and angle of rotation and it computes the resultant vector after rotation. Basic operation and concept of both the mode of operation is same that we rotate the vector with fixed finite angles but there is a small difference in the tracing of variables.

4.4.1 Rotation mode

If co-ordinate of input vector (x, y) and the angle of rotation (θ) is know then the algorithm computes the resultant vector (x_f, y_f), and final angle after rotation by θ. Equation 4.20, 21 and 22 will be same but next value of d will be given by Equation 4.28.

$$d(i+1)=\begin{cases} +1 & z(i)\geq 0 \\ -1 & z(i)<0 \end{cases} \tag{4.28}$$

In rotation mode z(0)= θ (angle of rotation). After completion of n rotations, we get the vector ($x(i_{final})$, $y(i_{final})$) with an error in magnitude and when we compensate this by multiplying with K then we get the final co-ordinate of the final vector (x_f, y_f).

4.4.2 Example of rotation mode

Suppose (x,y) is vector input where x=1 and y=0.125 and θ= 67^0 is the angle of rotation so x(0)=1, y(0)=0.125 and in radian z(0)=1.1693 rad.

In Table 4.4 after 12 iterations $x(i_{final})$=0.4531 and $y(i_{final})$=1.5965. For magnitude compensation the result need to be multiplied by K=0.607252935009. Then x_f=0.2753 , y_f=0.9693 and error <2^{-12}.

i	x(i)	y(i)	z(i)	d(i)
0	1.000000	0.125000	1.169371	1.000000
1	0.875000	1.125000	0.383972	1.000000
2	0.312500	1.562500	-0.079675	-1.000000
3	0.703125	1.484375	0.165303	1.000000
4	0.517578	1.572266	0.040948	1.000000
5	0.419312	1.604614	-0.021470	-1.000000
6	0.469456	1.591511	0.009770	1.000000
7	0.444588	1.598846	-0.005854	-1.000000
8	0.457079	1.595373	0.001958	1.000000
9	0.450847	1.597158	-0.001948	-1.000000
10	0.453967	1.596278	0.000005	1.000000
11	0.452408	1.596721	-0.000972	-1.000000
12	0.453188	1.596500	-0.000483	-1.000000
13	0.453577	1.596389	-0.000239	-1.000000

Table 4.4. Table for the example of rotation mode

4.4.3 Vectoring mode

In vectoring mode of operation, co-ordinate of input (x, y) is given and input vector is rotated by its initial angle to compute the magnitude of input vector and angle of rotation. Equation 4.20, 21 and 22 will be same but next value of d will be given by Equation 4.29.

$$d(i+1)=\begin{cases} -1 & y(i)\geq 0 \\ +1 & y(i)<0 \end{cases} \qquad (4.29)$$

and $z(0)=0$

The algorithm tries to make the angle of vector zero so that x co-ordinate itself represents the magnitude or absolute value of vector and y coordinate will be almost zero. Since in order to make the angle zero vector has to be rotated by its initial angle so finally variable z will contain the value of rotated angle which will be equal to the initial angle of the vector.

After completion of n rotations, we get the vector $(x(i_{final}), y(i_{final}))$ with an error in magnitude and when we compensate this by multiplying by K then we get the final coordinate of the final vector (x_f, y_f), given by Equation 4.30 and Equation 4.31 and z_f is given by Equation 4.32.

$$x_f=(x^2+y^2)^{0.5} \tag{4.30}$$

$$y_f \simeq 0 \tag{4.31}$$

$$z_f=\tan^{-1}(y/x) \tag{4.32}$$

i	x(i)	y(i)	z(i)	d(i)
0	0.430000	0.750000	0.000000	-1
1	1.180000	0.320000	0.785398	-1
2	1.340000	-0.270000	1.249046	1
3	1.407500	0.065000	1.004067	-1
4	1.415625	-0.110938	1.128422	1
5	1.422559	-0.022461	1.066003	1
6	1.423260	0.021994	1.034763	-1
7	1.423604	-0.000244	1.050387	1
8	1.423606	0.010877	1.042575	-1
9	1.423649	0.005317	1.046481	-1
10	1.423659	0.002536	1.048434	-1
11	1.423661	0.001146	1.049411	-1
12	1.423662	0.000451	1.049899	-1
13	1.423662	0.000103	1.050143	-1
14	1.423662	-0.000071	1.050265	1

Table 4.5. Table for the example of vectoring mode

4.4.4 Example of Vectoring mode

Suppose (x,y) is vector input where x=0.43 and y=0.75 and z(0)=0 in vectoring mode. So x(0)=1, y(0)=0.125 and z(0)=0.

In Table 4.5 after 12 iterations $x(i_{final})$=1.423662 and $y(i_{final})$=-0.00071 and z_f=1.050265. For magnitude compensation the result is multiplied by K=0.607252935009, then x_f=0.864522, y_f=0 and error $<2^{-12.}$

4.5 Flow chart for the control logic

Figure 4.4 is showing the Flow chart and with the help of this we can design FSM for CORDIC algorithm that will work as a complex multiplier as well as it will compute absolute value of a vector by setting the flag mul_abs.

This flow chart can be used for rotation based architecture where we can reuse the same hardware multiple times. Depending on the value of mul_abs flag we can select a particular operation as:

$$\text{mul\_abs} = \left\{ \begin{array}{ll} 0 & \text{Rotate the vector with a given angle (complex multiplier)} \\ 1 & \text{calculate absolute value and angle of the vector} \end{array} \right\}$$

Using 'mul_abs' input, proper functionality of CORDIC block can be selected. If just changes the mode of operation from vectoring to rotation mode.

Flow chart chows the iteration algorithm where a single processing unit can be reused to implement required number of stages.

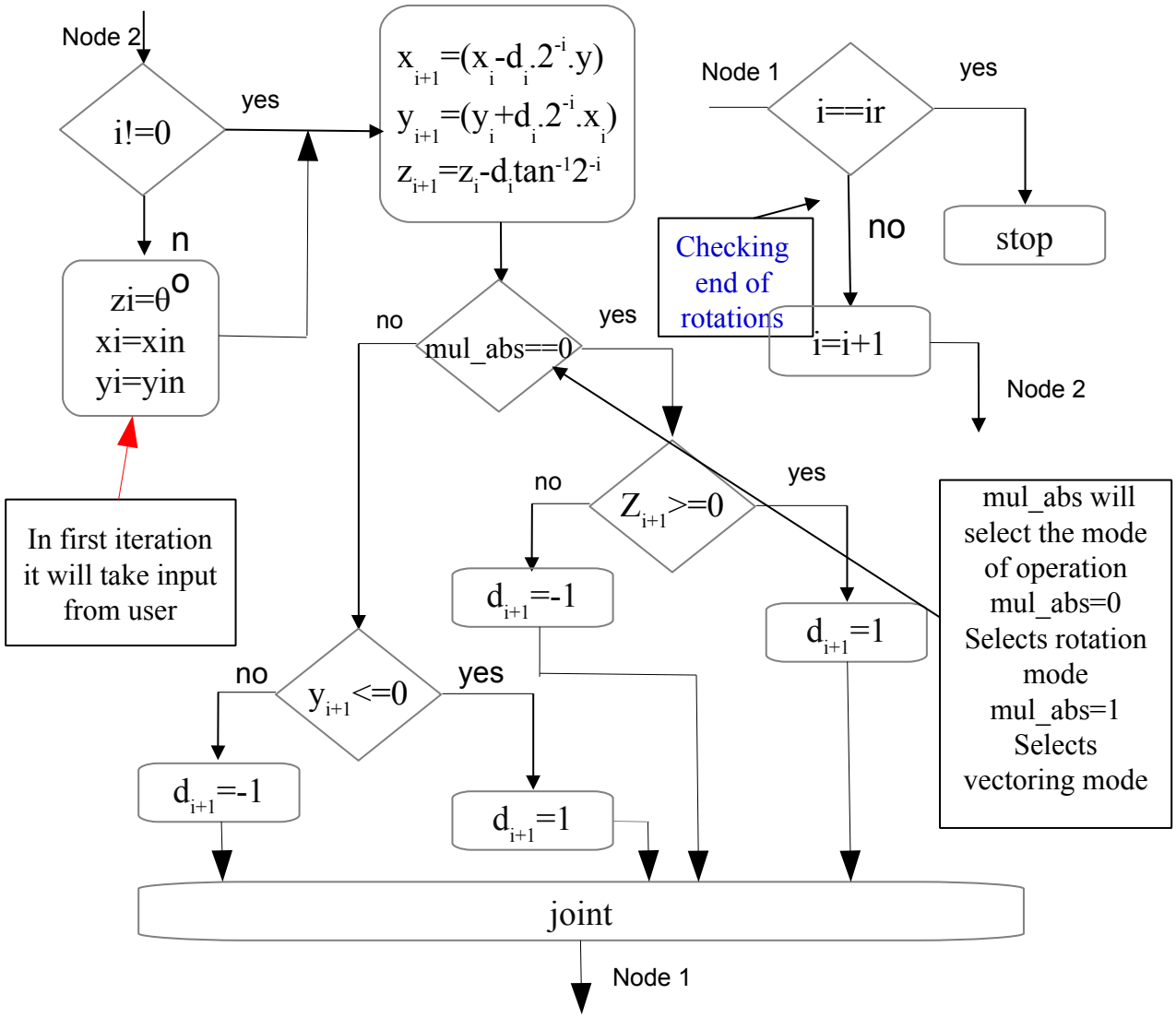

Figure 4.4. Flow chart for the control logic

4.6 Hardware estimation for complex multiplier and absolute value function

Hardware requirement is dependent on the architecture we are using. There are two basic architectures to implement the algorithm.

1. **Iteration based architecture**

In i^{th} iteration shifter will shift the input x and y by i-bit and after that they will be added or subtracted according to equation depending on control input (d(i) in the equation) and value of x(i+1) and y(i+1) is computed and stored in two registers. Control logic decides the control input based on mode of operation and comparison. In this case system will be slower but hardware requirement is less.

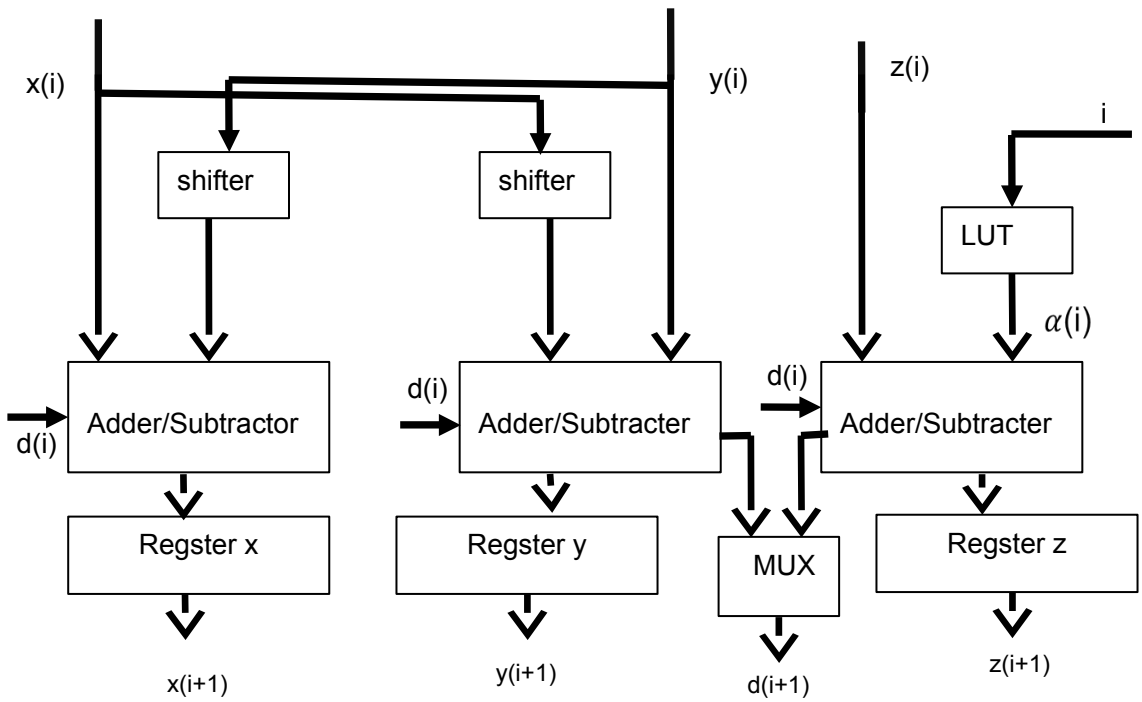

Figure 4.5. Basic hardware required for the algorithm

Figure 4.5 is showing the hardware for this architecture. It needs

1. Three adders / subtractors (based on control logic they can add or subtract the data)
2. Two data shifters (that can shift the data by i-bit where i is a variable)
3. One n-word LUT to store the fixed angle of rotation in each iteration
4. One control unit (that will iterate the data)
5. Registers to store the results after each iteration

2. Pipelined architecture

Here hardware requirement is almost n-times greater than rotation based architecture where n is

the number of iterations. But the system is faster because output of one stage can be directly applied to the next stage. There is no need to store the output and wait for the next clock.

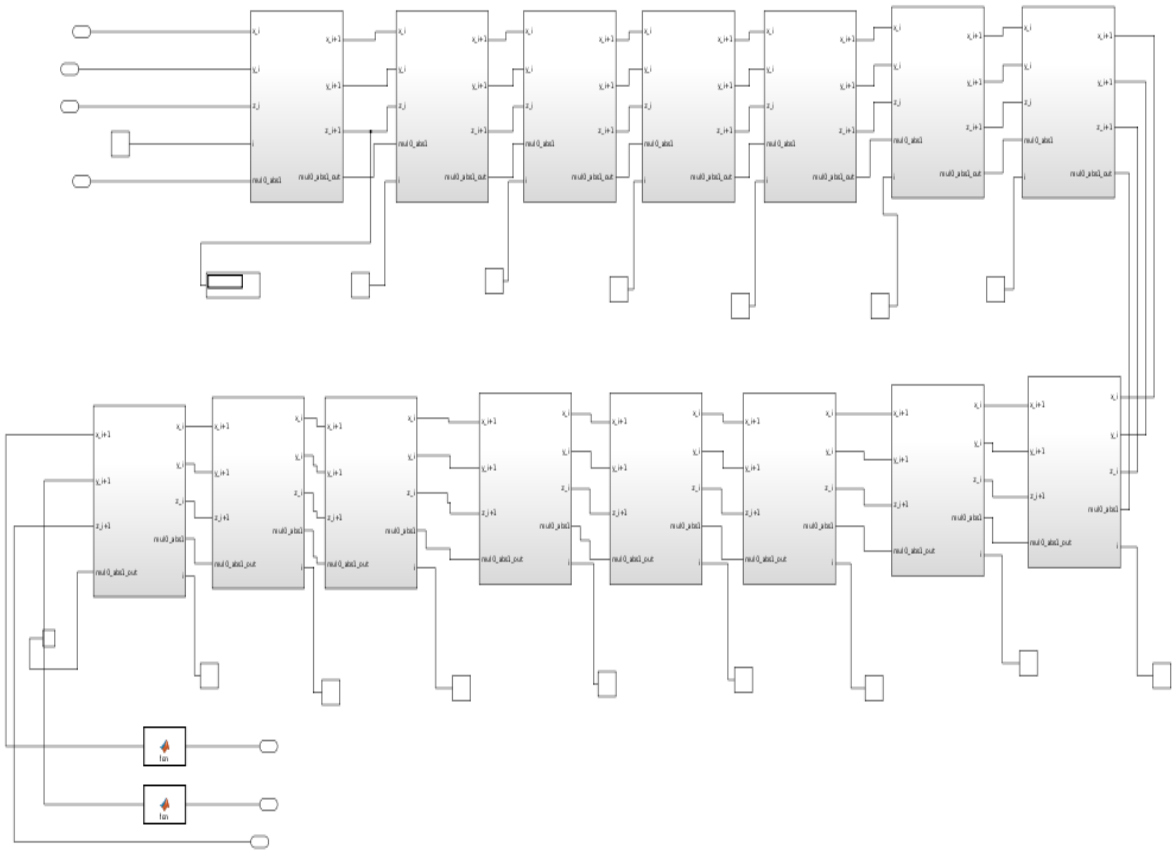

Figure 4.6 Pipelined architecture

n-stages of same hardware can be cascaded to make the system so that, it can directly take input from previous stage and can provide input to the next stage. Here we are avoiding the iterations and we are not using the same hardware for each stage but we are using one complete set of hardwares shown in the Figure 4.5 for each stage.

Figure 4.6 is showing the implementation of 14 stage pipelined architecture where each block is representing the hardware shown in Figure 4.5 except that in place of LUT here we will use a register that will store rotation angle corresponding to that stage.

4.7 Convergence issue and precision in circular co-ordinate system

The algorithm will be convergent if it satisfies Equation 4.33.

$$|z(i)| \leqslant \sum_{j=i+1}^{\infty} \tan^{-1}(2^{-j}) \tag{4.33}$$

So maximum angle through which the algorithm can rotate a vector is given by Equation 4.34 as

$$\theta_{max} = z[0]_{max} = \sum_{j=0}^{\infty} \tan^{-1}(2^{-j}) \approx 1.7429(99.88^0) \tag{4.34}$$

Since the difference of angle between target vector and rotating vector at i^{th} stage is given Equation 4.20 and we can in Table 4.2 that for i>10

$\tan^{-1}2^{-i} \approx 2^{-i}$ (with an error of order 2^{-10})

So after each iteration error will reduce by 2^{-i} and we can make the number of iterations finite by choosing an appropriate error limit.

If we are using a 14 bit ADC and we need an accuracy of $0.1*2^{-13}$ (0.1*step width) then to achieve this accuracy, we have to go for $\log_2(0.1*2^{-13}) = 16.3$ or 17 iterations but for 17 iterations we need at least 18 bit registers to store values of x, y and z so that even if we multiplying the data by 2^{-17} there is some data left.

For an accuracy of 2^{-14} (0.5*step width), we need 14-stages and at least 15-bit registers to store values of x, y and z after each iteration.

4.8 Extending the range of input angle in circular co-ordinate system

If we want to rotate the vector (x, y) by an angle outside the range of conventional algorithm, we need to extend the range of the algorithm. It can be done by adding one more stage before the start of iterations that can rotate the vector by $\pm 90^0$, if rotation angle z(0) is out side the range or angle of rotation, means it is greater than 90^0 or less than -90^0. Rotating the vector by 90^0 is nothing but

changing sign of x and y co-ordinates.

We can show this process using a flow chart described in Figure 4.7. So some extra hardware is required to convert the vector (x, y) to (x_{new}, y_{new}) and difference of angle will now be changed by $\pm 90^0$.

4.9 Pros and cons of both architectures

1. Due to multiple iterations processing time will be higher in iteration based architecture.
2. In pipelined architecture hardware requirement will be higher.
3. For n-number of iterations at least n+1 bit registers will be required to store values of x, y and z after each iteration.

4.10 Total hardware requirement and issues

For 14 bit ADC and assuming that error limit is $0.1*2^{-13}$ (0.1*step width), for this we need $\log_2(0.1*2^{-13})=16.3$ or minimum 17 iterations including iteration for i=0 we have to implement 18-stages and for that we need

1. 54 adders / subtractors
2. 34 shifters (No need of shifters for i=0)
3. 18 words register to store the values of $\tan^{-1}2^{-i}$
4. 18 comparators (sgn function)
5. One range extension block

But using 14-bit registers we can not iterate it for more than 13-iterations as in 14th iteration, when x and y will be multiplied by 2^{-14} then it will result in zero and adding or subtracting zero will not change the the value of x and y for next iteration and finally we will get an incorrect result.

So, for n-iterations we have to take at least n+1 bit registers to store values of x, y and z or we

can fix our accuracy requirement according to the available register width.But using 14-bit registers we can not iterate it for more than 13-iterations as in 14th iteration, when x and y will be multiplied by 2^{-14} then it will result in zero and adding or subtracting zero will not change the the value of x and y for next iteration and finally we will get an incorrect result.

So, for n-iterations we have to take at least n+1 bit registers to store values of x, y and z or we can fix our accuracy requirement according to the available register width.

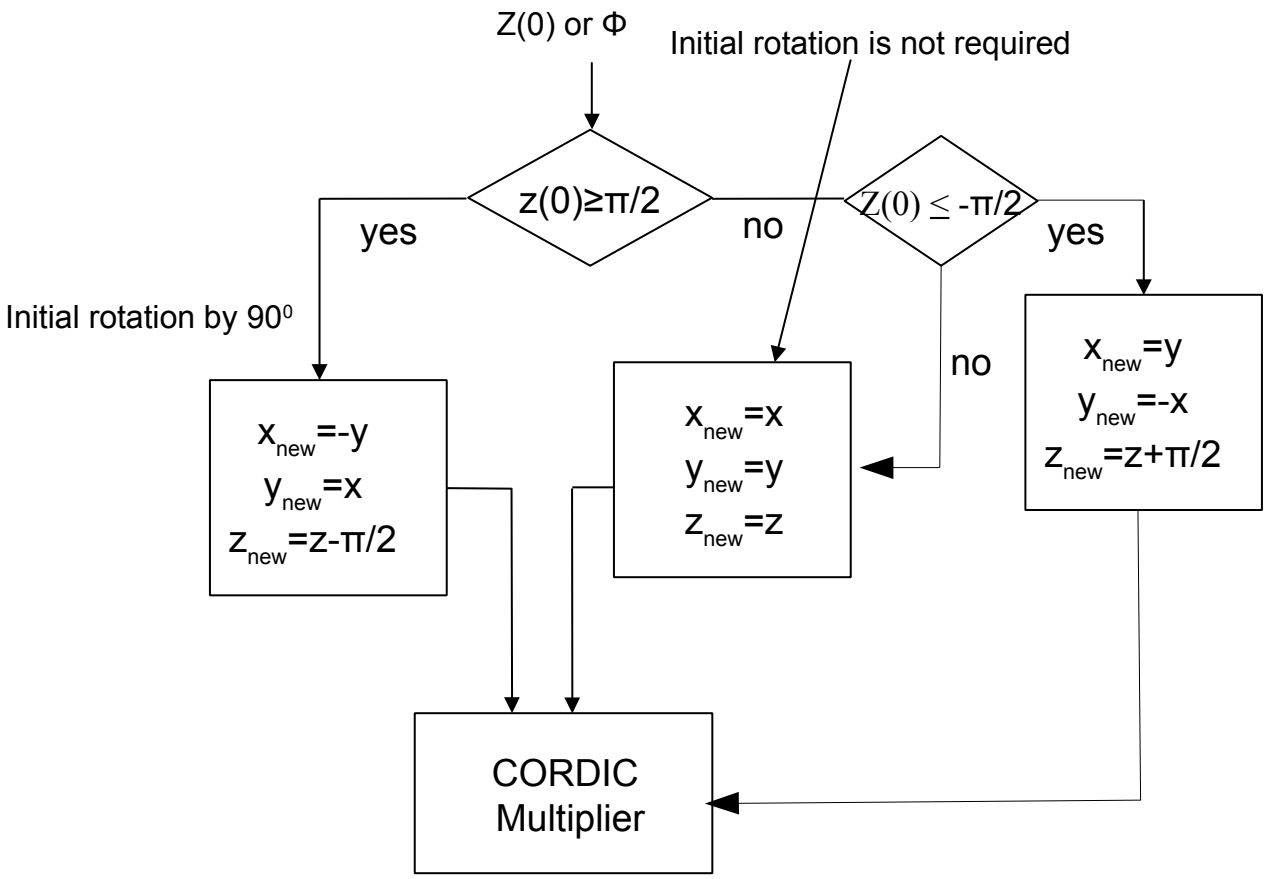

Figure 4.7. Flow chart to extend the range of algorithm

For one radix-4 butterfly, we will need 3 such complex multipliers (each multiplier will be implemented by CORDIC algorithm). So, the hardware requirement will be 162 adders / subtractors,

102 shifters, 54 comparators and 18 registers to store the values of $\tan^{-1}2^{-i}$.

It will save N (2^16) word ROM and need of 3 complex multipliers (12 real multipliers and 6 real adders) in a radix-4 butterfly but it can increase the processing time of the butterfly unit.

4.11 CORDIC algorithm for the implementation of logarithmic function

In order to compute the circular functions like sin, cos and tan we rotate the vector on a circular path by an angle of θ and if rotating vector coordinate is (x, y) then rotated vector coordinate (x', y') will be given by Equation 4.35 and 36.

$$x' = x.\cos\theta - y.\sin\theta \tag{4.35}$$

and

$$y' = x.\sin\theta + y.\cos\theta \tag{4.36}$$

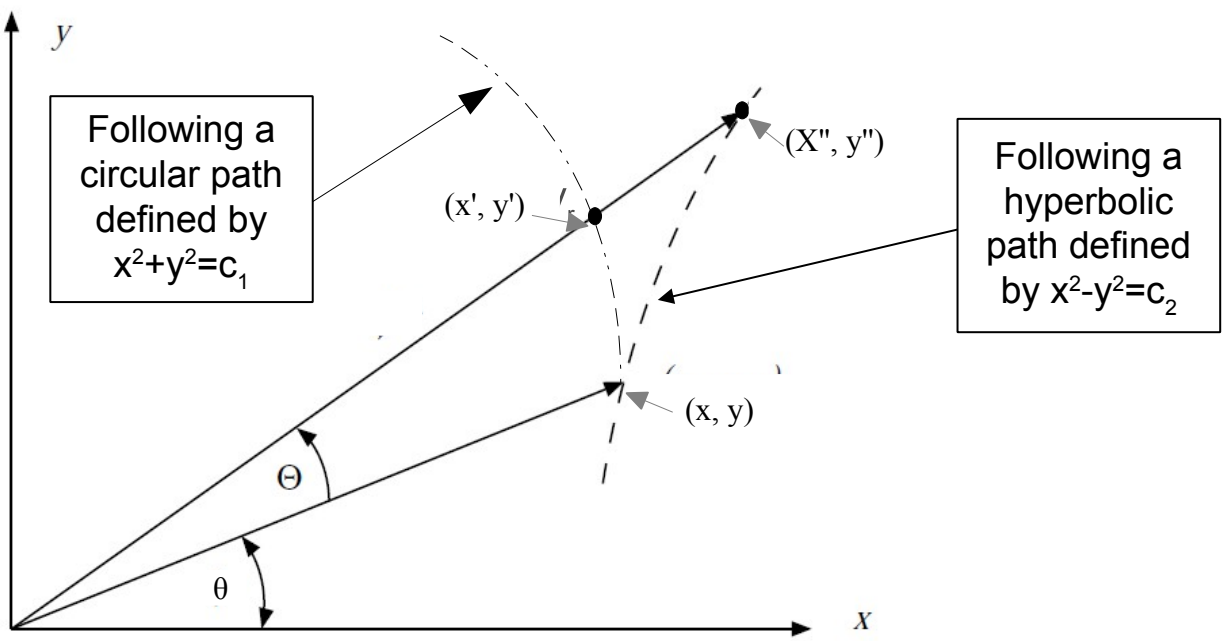

Figure 4.8 Rotation in hyperbolic as well as circular coordinate system

But if we rotate the vector by following a hyperbolic path in place of circular path then if rotating vector coordinates are (x, y) and rotated vector coordinates are (x', y') then it will be given by

Equation 4.37 and 38.

$$x''=x.\cosh\theta + y.\sinh\theta \tag{4.37}$$

and

$$y''=x.\sinh\theta + y.\cosh\theta \tag{4.38}$$

As we can see in Figure 4.7 that if vector having coordinate (x, y) is rotated by following a circular path centered at origin and having constant radius ($x^2+y^2=c_1$) then new coordinate of rotated vector is (x', y') and if it is rotated by following hyperbolic path (rectangular hyperbola which equation is $x^2-y^2=c_2$) then the new co-ordinate of rotated vector is given by (x", y").

Equation 4.37 and 38 can be rearranged as given by the Equation 4.39 and 40.

$$x''=\cosh\theta\ (x + y*\tanh\theta) \tag{4.39}$$

$$y''=\cosh\theta\ (y + x*\tanh\theta) \tag{4.40}$$

Angle corresponding to new vector (x", y") will be given by Equation 4.41.

$$\theta''=\tan^{-1}\frac{(y+x*tanh\theta)}{(x+y*tanh\theta)} \tag{4.41}$$

So tanhθ is responsible for the change in angle of the rotated resultant vector and coshθ is just changing the magnitude of the resultant vector.

Now direct computation of x" and y" needs the value of tanhθ and coshθ for any arbitrary angle θ and also it needs multipliers and adders.

Using CORDIC algorithm we can rotate the vector having coordinate (x, y) in hyperbolic co-ordinate system by breaking the arbitrary θ angle into a series of fixed angles such that algebraic sum of the series will be equal to θ as given by Equation 4.18.

Here we can take any series that satisfies the criteria written in Equation 4.42

$$\left|\frac{\theta_i}{\theta_{i+1}}\right| \leq 2 \tag{4.42}$$

Now in order to replace the multiplier by shifters we will take θ_i as given by Equation 4.43 and so that we can replace $\tan\theta_i$ according to Equation 4.44.

$$\theta_i = \pm \tanh^{-1}(2^{-i}) \tag{4.43}$$

so

$$\tanh\theta_i = \pm 2^{-i} \tag{4.44}$$

We can take another series that satisfies that condition but for other series we need to multiply the value of $\tanh\theta_i$ to x and y, for that we need multipliers.

Now in place of rotating the vector with an arbitrary angle θ we will rotate the vector with fixed angles θ_i (for i^{th} rotation or iteration) according to the series given in Equation 4.43. Now after i^{th} iteration new coordinates of the vector will be given by Equation 4.45 and 46.

$$x(i+1)=\cosh\theta_i\,[x(i) + d_{i.}\, y(i).2^{-i}] \tag{4.45}$$

$$y(i+1)=\cosh\theta_i\,[y(i) + d_{i.}x(i).2^{-i}] \tag{4.46}$$

We can define a variable z to trace the difference between target vector and rotating vector and it can be given by Equation 4.47.

$$z(i+1) = z(i) - d_i * \tanh^{-1}(2^{-i}) \tag{4.47}$$

where $di = +1$

i	2^-i	$\tanh^{-1}(2^{-i})$	$\log_2(\tanh^{-1}(2^{-i}))$
1	0.500000	0.549306	-0.86
2	0.250000	0.255413	-1.97
3	0.125000	0.125657	-2.99
4	0.062500	0.062582	-4.00
5	0.031250	0.031260	-5.00
6	0.015625	0.015626	-6.00
7	0.007813	0.007813	-7.00
8	0.003906	0.003906	-8.00
9	0.001953	0.001953	-9.00
10	0.000977	0.000977	-10.00
11	0.000488	0.000488	-11.00
12	0.000244	0.000244	-12.00
13	0.000122	0.000122	-13.00

Table 4.6 Error after each iteration ($\tanh^{-1}2^{-i}$)

So if we will go for infinite number of rotations then the difference will be definitely zero but for finite number of stages, at each stage angle difference will be given by Equation 4.48

$$z(i+1) - z(i) = -\theta_i = -d_i * \tanh^{-1}2^{-i} \tag{4.48}$$

So after i^{th} iteration error will be $\tanh^{-1}2^{-i}$

We can see from the Table 4.6 that error is almost 2^{-i} for $i \geq 4$ and we can not iterate for i=0 using this series of θ_i because $\tanh^{-1}1$ is undefined.

Again we can leave the multiplication by magnitude correction factor $\cosh\theta_i$ at each stage and we can directly multiply the final result by a magnitude correction factor K given by Equation 4.49 and using Equation 4.50 and 51 we can calculate the value of K.

$$K = \cosh\theta_0 * \cosh\theta_1 * \cosh\theta_2 * \cosh\theta_3 * \ldots * \cosh\theta_\infty \tag{4.49}$$

where

$$\cosh\theta_i = \frac{1}{(1-\tanh^2\theta_i)^{\frac{1}{2}}} \tag{4.50}$$

so for $\tanh\theta_i = \pm 2^{-i}$

$$cosh\theta_i = \frac{1}{(1-2^{-2i})^{\frac{1}{2}}} \tag{4.51}$$

So we can directly calculated the value of K and can store it into a register and after completion of all the iterations we can multiply the result with K to get the final resultant vector.

Here we can notice one point that the value of $\cosh\theta_i$ is greater than one for $i \geq 1$, so after each rotation magnitude of new vector will shrink if we do not multiply with $\cosh\theta_i$ in each iteration.

So leaving this factor modified equation will be given by Equation 4.52, 4.53, and 4.54.

$$x(i+1)=[x(i) + d_i*y(i)*2^{-i}] \tag{4.52}$$

$$y(i+1)=[y(i) + d_i*x(i)*2^{-i}] \tag{4.53}$$

$$z(i+1) = z(i) - d_i*\tanh^{-1}(2^{-i}) \tag{4.54}$$

4.12 Convergence issue and precision for hyperbolic co-ordinate system

In order to ensure the convergence for the sequence of angels $\tanh^{-1}(2^{-i})$ it should satisfy the Equation 4.55.

$$|z(i)| \leqslant \sum_{j=i+1}^{\infty} \tanh^{-1}(2^{-j}) \tag{4.55}$$

but for this sequence, it does not satisfy convergence criteria written in Equation 4.55 and for this sequence, the value of z(i) is given by Equation 4.56.

$$|z(i)| > \sum_{j=i+1}^{\infty} \tanh^{-1}(2^{-j}) \tag{4.56}$$

so algorithm will not converge for given sequence of angles. In order to ensure the convergence we need to increase the value of right hand side for the Equation 4.55 . Result shows that if we repeat

few iterations according to series given by Equation 4.57

$$\sum_{j=i+1}^{\infty} \tanh^{-1}(2^{-j}) < |z(i)| < \sum_{j=i+1}^{\infty} \tanh^{-1}(2^{-j}) + \tanh^{-1}(2^{-(3i+1)}) \tag{4.57}$$

Since $\tanh^{-1}(1)$ is undefined, so rotation will start from i=1.

According to the results if we repeat the iterations 4, 13, 40, ….. k, 3k+1, ….. then algorithm will converge. Now we need to rotate the vector for i=1, 2, 3, 4, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 13, 14.....

So maximum angle through which the algorithm can rotate a vector in hyperbolic coordinate system is given by Equation 4.58.

$$\theta_{max} = z[0]_{max} = \sum_{j=0}^{\infty} \tanh^{-1}(2^{-j}) + \tanh^{-1}(2^{-4}) + \tanh^{-1}(2^{-13}) + ... \approx 1.11817 \tag{4.58}$$

Since the difference of angle between target vector and rotated vector at i^{th} stage is z(i) and it is given by Equation 4.48. For infinite number of rotations the difference will be zero but if we are iterating for finite number of iterations then after each iteration error will be $\tanh^{-1}2^{-i}$.

We can see in the Table 4.7 that for i>10 error is less than 10^{-10}, so we can say that for i>10

$$\tanh^{-1}2^{-i} \approx 2^{-i}$$

So after each iteration error in angle will reduce by 2^{-i} and we can make the number of iterations finite by choosing an appropriate error limit.

i	2^-i	$\tanh^{-1}(2^{-i})$	Error
0	1.000000	--	
1	0.500000	0.549306	4.93E-002
2	0.250000	0.255413	5.41E-003
3	0.125000	0.125657	6.57E-004
4	0.062500	0.062582	8.16E-005
5	0.031250	0.031260	1.02E-005
6	0.015625	0.015626	1.27E-006
7	0.007813	0.007813	1.59E-007
8	0.003906	0.003906	1.99E-008
9	0.001953	0.001953	2.48E-009
10	0.000977	0.000977	3.10E-010
11	0.000488	0.000488	3.88E-011
12	0.000244	0.000244	4.85E-012
13	0.000122	0.000122	6.06E-013

Table 4.7 Reduction in error in each stage

Now if we calculate magnitude compensation factor K for the new value of index i (including the repetition) then K=1.207534495276374. Since K>1 so it will increase the magnitude of resultant vector.

4.13 Mode of operation in hyperbolic co-ordinate system

Again in hyperbolic coordinate system also, vector can be rotated in two different modes, known as the "Rotation mode" and the "Vectoring" mode.

4.13.1 Rotation mode

In the rotation mode, the co-ordinate components of a vector (x, y) and an angle of rotation (θ) are given and the co-ordinate components of the final vector (x_f, y_f) are computed. Equations 4.52, 4.53 and 4.54 will remain same but next value of variable d will be given by Equation 4.59.

$$d(i+1)=\begin{cases} +1 & z(i)\geq 0 \\ -1 & z(i)<0 \end{cases} \tag{4.59}$$

Here in rotation mode z(0)= θ (angle of rotation). After completion of n rotation we get the

vector (x', y') with an error in magnitude and when we compensate this by multiplying with K then we get the coordinate of final vector (x_f, y_f).

4.13.2 Vectoring mode

In the vectoring mode, the co-ordinate components of a vector (x, y) are given and the magnitude and angular argument of the final vector are computed. Equations -52, 53 and 54 will remain same but next value of variable d will be given by Equation 4.60.

But here

$$d(i+1)=\begin{cases} -1 & y(i)\geq 0 \\ +1 & y(i)<0 \end{cases} \tag{4.60}$$

and $z(0)=0$

Here the algorithm tries to make the angle of vector zero so that, x co-ordinate itself represents the magnitude of the vector in hyperbolic co-ordinate system and y co-ordinate will be almost zero. Since in order to make the angle zero vector has to rotate by its initial angle, so finally z variable will have the rotated angle which will be the initial angle of the vector.

After completion of n rotations, we get the vector ($x(i_{final})$, $y(i_{final})$) with an error in magnitude and when we compensate this by multiplying by K then we get the final coordinate of the final vector (x_f, y_f), given by Equation 4.61 and Equation 4.62 and z_f is given by Equation 4.63.

$$x_f=(x^2-y^2)^{0.5} \tag{4.61}$$

$$y_f \simeq 0 \tag{4.62}$$

$$z_f=\tanh^{-1}(y/x) \tag{4.63}$$

4.14 Manipulation of input data to implement Logarithmic function

As we know the identity shown in Equation 4.64

$$\ln(p) = 2 * \tanh^{-1}\left(\frac{p-1}{p+1}\right) \tag{4.64}$$

So if we replace the initial input (co-ordinates of vector x and y) in such a way that

x=p+1 and y=p-1

Then for this manipulation

$\ln(p)=2*z_f$

We can change the base of logarithmic function from 'e' to '10' by dividing the result by ln(10).

4.14.1 Example of implementation of logarithmic function

Suppose we want to calculate the value of $\log_{10}2$ using CORDIC algorithm then coordinate of input vector will be

x=2+1=3

y=2-1=1

z(0)=0

So for iterations x(0)=3, y(0)=1 and z(0)=0 and we will start the iterations from i=1.

i	x(i)	y(i)	z(i)	d(i)
0	3.000000	1.000000	0.00000000	-1
1	2.500000	-0.500000	0.54931000	1
2	2.375000	0.125000	0.29389000	-1
3	2.359400	-0.171880	0.41955000	1
4	2.348600	-0.024414	0.35697000	1
4	2.347100	0.122380	0.29447000	-1
5	2.343300	0.049028	0.32573000	-1
6	2.342500	0.012415	0.34136000	-1
7	2.342400	-0.005886	0.34917000	1
8	2.342400	0.003264	0.34526000	-1
9	2.342400	-0.001311	0.34721000	1
10	2.342400	0.000976	0.34624000	-1
11	2.342400	-0.000167	0.34673000	1
12	2.342400	0.000404	0.34648000	-1
13	2.342400	0.000118	0.34660000	-1
13	2.342400	-0.000167	0.34673000	1
14	2.342400	-0.000025	0.34667000	1

Table 4.8 Vector co-ordinate after each iteration (for logarithmic function)

We can see in the Table 4.8 that after 14 iterations z_f=0.346670

So ln(2) = 2*0.346670 = 0.67334000

We can change the base of logarithmic function by dividing it by ln(10). So the final result will be:

$\log_{10}(2)$=ln(2) / ln(10) =0.67334000 /2.3025851=0.301113736474338

The error is less than 2^-13.54 (by manual calculations).

4.15 Manipulation of input data to implement square root function

In order to implement square root function we can use an identity given by Equation 4.65.

$$2*\sqrt{a*p}=\sqrt{(p+a)^2-(p-a)^2} \tag{4.65}$$

Now we can use Equation 4.61 to implement the square-root function by using the identity written in Equation 4.65. By assuming that 'p' is the input argument for the square-root function and 'a' is a constant, we can change the value of 'a' according to our convenience in data manipulation but it will affect the input range for square root function.

So if we replace the initial input (co-ordinates of input vector (x, y)) in such a way that

$$a=0.25,\ x=p+0.25 \text{ and } y=p-0.25$$

then for this manipulation

$$x'=x_f / K= \sqrt{p} / K$$

we can do another manipulation as

$$a=1,\ x=p+1 \text{ and } y=p-1$$

for this manipulation

$$x'=2*x_f / K= 2*\sqrt{p} / K$$

And using this manipulation we can utilize same hardware for logarithmic function as well as square-root function but the input range will shift (we will discuss the range in next section).

Where K is magnitude compensation factor and finally we need to multiply x' by K(for first the manipulation and K/2 for second manipulation) to get the required result.

4.15.1 Example of implementation of square root function

Suppose we want to calculate the value of √2 using CORDIC algorithm then co-ordinates of the input vector will be

$$x=2+0.25=2.25$$

$$y=2-0.25=1.75$$

z(0)=0

So for iterations x(0)=2.25, y(0)=1.75 and z(0)=0 and we will start the iterations for i=1.

i	x(i)	y(i)	z(i)	d(i)
0	2.250000	1.750000	0.000000	-1
1	1.375000	0.625000	0.549310	-1
2	1.218800	0.281250	0.804720	-1
3	1.183600	0.128910	0.930380	-1
4	1.175500	0.054932	0.992960	-1
4	1.172100	-0.018539	1.055500	1
5	1.171500	0.018089	1.024200	-1
6	1.171200	-0.000216	1.039800	1
7	1.171200	0.008934	1.032000	-1
8	1.171200	0.004359	1.035900	-1
9	1.171200	0.002071	1.037900	-1
10	1.171200	0.000928	1.038800	-1
11	1.171200	0.000356	1.039300	-1
12	1.171200	0.000070	1.039600	-1
13	1.171200	-0.000073	1.039700	1
13	1.171200	0.000070	1.039600	-1
14	1.171200	-0.000002	1.039600	1

Table 4.9 Vector co-ordinate after each iteration (for square root function)

We can see in Table 4.9 that after 14 iterations x'=1.1712 and K=1.207534495276374

So x_f= √2 =1.1712*1.207534495276374 = 1.414264400867689 and error is ~2^-14.26 (by manual calculations)

4.16 Input range for conventional algorithm and extension of input range

Input range can be defined by condition under which the algorithm will be convergent. We have seen that the algorithm will be convergent if it satisfies the Equation 4.57.

According to results if we repeat the iterations 4, 13, 40, ….. k, 3k+1, …. then algorithm will be convergent. Now we need to rotate the vector for i=1, 2, 3, 4, **4**, 5, 6, 7, 8, 9, 10, 11, 12, 13, **13**, 14..... and so on.

So maximum angle through which the algorithm can rotate a vector in hyperbolic co-ordinate system is given by Equation 4.58. According to that θ_{max} is 1.11817 radian and in vectoring mode θ is given by Equation 4.66.

$$\theta=\tanh^{-1}(y/x) = z_f \tag{4.66}$$

Since in vectoring mode z(0)=0, so maximum value of z_f will be the θ_{max}. While θ will depend on the values of x and y, which can be more than θ_{max}. So input range extension will be required.

4.16.1 Input range for logarithmic function

Using Equation 4.66 maximum value $\tanh^{-1}(y/x)$ will be given by Equation 4.67 and Equation 4.68 is showing the maximum value of y/x.

$$|\tanh^{-1}(\frac{y}{x})|_{max} = 1.11817 \tag{4.67}$$

So

$$|\frac{y}{x}|_{max} = 0.80693 \tag{4.68}$$

For implementation of log(p), x=p+1, y=p-1 and $\ln(p)=2*\tanh^{-1}(y/x)$ where p is the input to logarithmic function. By putting the values of x and y in Equation 4.68, we get Equation 4.69.

$$|\frac{(p-1)}{(p+1)}|_{max} = 0.80693 \tag{4.69}$$

So p_{max} =9.35893

Magnitude of maximum angle in negative direction will also be given by Equation 4.58 but with a negative sign. So θ_{min}= -1.11817 (sum of all the angles with negative sign) and for this value of θ_{min}, using Equation 4.68 we can get Equation 4.70.

$$|\frac{(p-1)}{(p+1)}|_{min} = -0.80693 \tag{4.70}$$

It will give the minimum value of p as p_{min} =0.10685. So the input range is given by Equation 4.71.

$$0.10685 \leq p \leq 9.35893 \tag{4.71}$$

4.16.2 Input range for square root function

If we are manipulating the data for a=1, then the input range for square-root function will be same as for logarithmic function because manipulation of input data is same for both the functions. Input range will be different, if we are manipulating the data for a different value of 'a'.

For the implementation of square root function x=p+a, y=p-a where 'p' is the input to square-root function and 'a' is the constant. For this input, input angle will be given by Equation 4.72.

$$\theta = \tan^{-1}((p-a)/(p+a)) \tag{4.72}$$

Using Equation 4.58, θ_{max}= 1.11817, θ_{min}= -1.11817

So $$|\frac{(p-a)}{(p+a)}|_{max} = 0.80693$$

and $$|\frac{(p-a)}{(p+a)}|_{min} = -0.80693$$

It will give the input range as

$$0.10685 * a \leq p \leq 9.35893 * a$$

For a=1, input range for logarithmic function will be same as for square-root function but for a=0.25, both the maximum value as well as the minimum value will reduce by ¼. So the range is different for both the manipulation. For a=0.25 the input range will be

$$0.026713 \leq p \leq 2.339733$$

As we can see that input range is not enough for the practical purpose. So we need to increase the input range which can be done by increasing the maximum value of θ.

4.17 Extension of input range

If we see the Equation 4.58, we can increase the value of θ_{max} by increasing the number of terms in series but we have already summed up the series for zero to infinity (for i=0 to ∞).

We can increase the value of θ_{max} by extending the series for negative index as it will increase the value of right hand side of Equation 4.58. But for the series ' $\theta_i = \tanh^{-1} 2^{-i}$ ', we can not go for negative indexes because for $i \leq 0$, $\tanh^{-1} 2^{-i}$ is a complex number.

But we can choose another series that satisfies criteria written in Equation 4.42. Different research papers have proposed different series for negative index that satisfies this criteria and using those series (for negative indexes), we can increase the maximum value of θ.

4.17.1 Series-1

For negative index we can define θ_i, given by Equation 4.73.

$$\theta_i = \pm \tanh^{-1} * [1 - 2^{(i-2)}] \qquad \text{for } i=-M, -M+1, ...-1, 0 \tag{4.73}$$

And for this series $\tanh \theta_i$ is given by Equation 4.74.

$$\tanh \theta_i = 1 - 2^{(i-2)} \tag{4.74}$$

So for i>0 Equation 4.52, 4.53, 4.54 will remain same but for $i \leq 0$ it will be Equation 4.75, 4.76 and 4.77.

$$x(i+1) = x(i) + d_i * y(i) * (1 - 2^{i-2}) \tag{4.75}$$

$$y(i+1)=y(i)+d_i*x(i)*(1-2^{i-2}) \tag{4.76}$$

$$z(i+1)=z(i)-d_i*\tanh^{-1}(1-2^{i-2}) \tag{4.77}$$

We can see in Table 4.10, as we increase the number of iteration towards negative side our input range is increasing for logarithmic and square-root functions. For a=1, the value of p_{max} and p_{min} for square-root function will be same as for logarithmic function.

i_{min}	K	θmax	Logarithmic Function		Square-root Function(for a=0.25)	
			p_{max}	p_{min}	p_{max}	p_{min}
-6	22522	15.55	3.18E+013	3.15E-014	7.94E+012	7.87E-015
-5	1988.7	12.43	6.22E+010	1.61E-011	1.55E+010	4.02E-012
-4	248.11	9.66	2.44E+008	4.10E-009	6.09E+007	1.03E-009
-3	43.69	7.23	1.92E+006	5.21E-007	4.80E+005	1.30E-007
-2	10.84	5.16	30464	3.28E-005	7615.9	8.21E-006
-1	3.77	3.45	982.7	0.001018	245.68	0.000254

Table 4.10 Different input range for different value of i_{min} using series -1

So for by adding 7 more stages (from i= - 6 to i=0), we can increase the upper range up to 3.18e+13 for logarithmic function and up to 7.94e+12 for square-root function.

4.17.2 Series-2

We can define another series that satisfies the criteria written in Equation 4.42 and for this series θ_i and $\tan\theta_i$ will be given by Equation 4.78 and 4.79.

$$\theta_i=\pm\tanh^{-1}(1-2^{-2^{(-i+1)}}) \tag{4.78}$$

For this series

$$\tanh\theta_i=\pm(1-2^{-2^{(-i+1)}}) \tag{4.79}$$

For i>0 Equation 4.52, 4.53, 4.54 will remain same but for i ≤ 0 it will be Equation 4.80, 4.81 and 4.82

$$x(i+1)=x(i)+d_i*y(i)*(1-2^{-2^{(-i+1)}}) \tag{4.80}$$

$$y(i+1)=y(i)+d_i*x(i)*(1-2^{-2^{(-i+1)}}) \tag{4.81}$$

$$z(i+1)=z(i)-d_i*\tanh^{-1}(1-2^{-2^{(-i+1)}}) \tag{4.82}$$

We can see in Table 4.11 that as we are increasing the number of iteration towards negative side, θ_{max} as well as our input range is increasing rapidly for both the functions. For a=1, the value of p_{max} and p_{min} for square-root function will be same as for logarithmic function.

i_{min}	K	θ_{max}	Logarithmic Function		Square-root Function (for a=0.25)	
			p_{max}	p_{min}	p_{max}	p_{min}
-4	4.98E+008	24.26	~Inf	~0	~Inf	~0
-3	10755	12.82	1.36E+011	7.35E-012	3.40E+010	1.84E-012
-2	59.41	6.93	1.04E+006	9.64E-007	2.59E+005	2.41E-007
-1	5.25	3.81	2030.9	0.000492	507.73	0.000123

Table 4.11 Different input range for different value of i_{min} using series-2

We can see that by adding 5 more stages (from i=- 4 to i=0) we can increase our input range to a very large value. But in this series value of K is very high and we need a register that can store this value and a multiplier corresponding to that accuracy to get the final output.

14.17.3 Hardware requirement to implement the series for negative index (for i<0)

We can see in Figure 4.9 that for the implementation of series-1, we need 5 adders/subtractors, 2 shifters and one register to store the fixed angle .

Figure 4.10 shows the hardware required for the implementation of series-2. We need 5 adders/subtractors, 3 shifters and one register to store the fixed angle. We can observe that it needs one more shifter to implement the series-2. Although the series-2 needs one more shifter but the series

needs lesser number of stages for the same range.

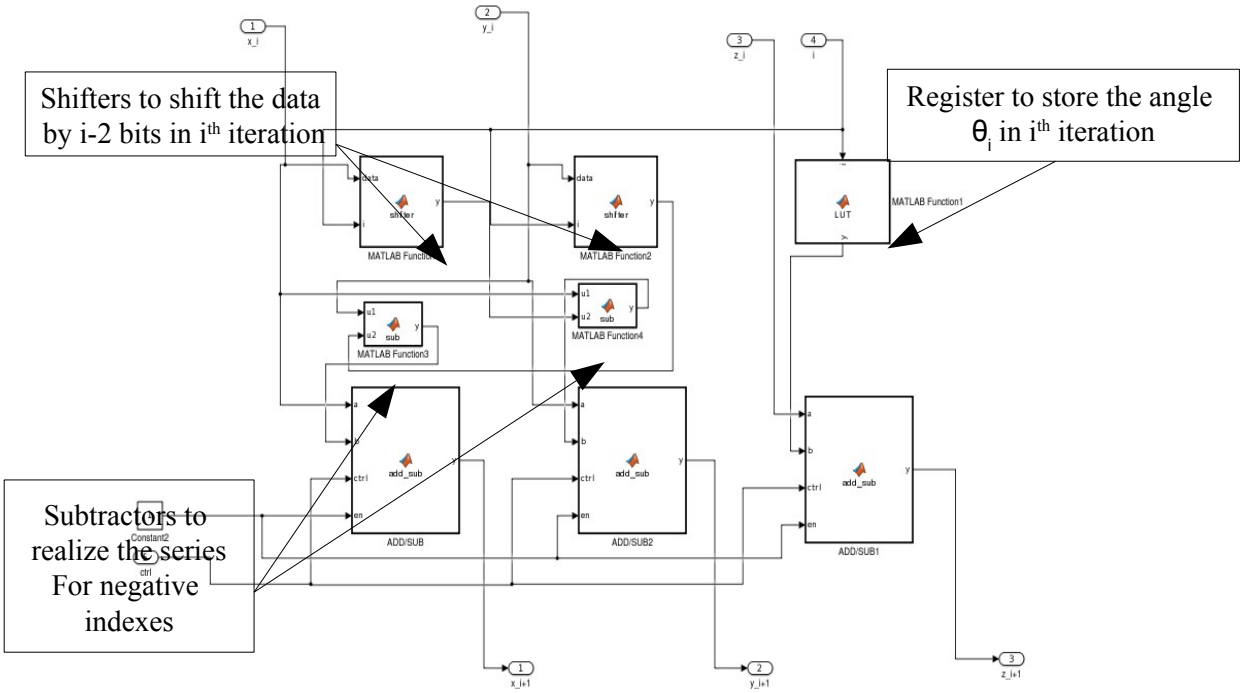

Figure 4.9 Hardware required to implement one stage for i≤0 using series-1

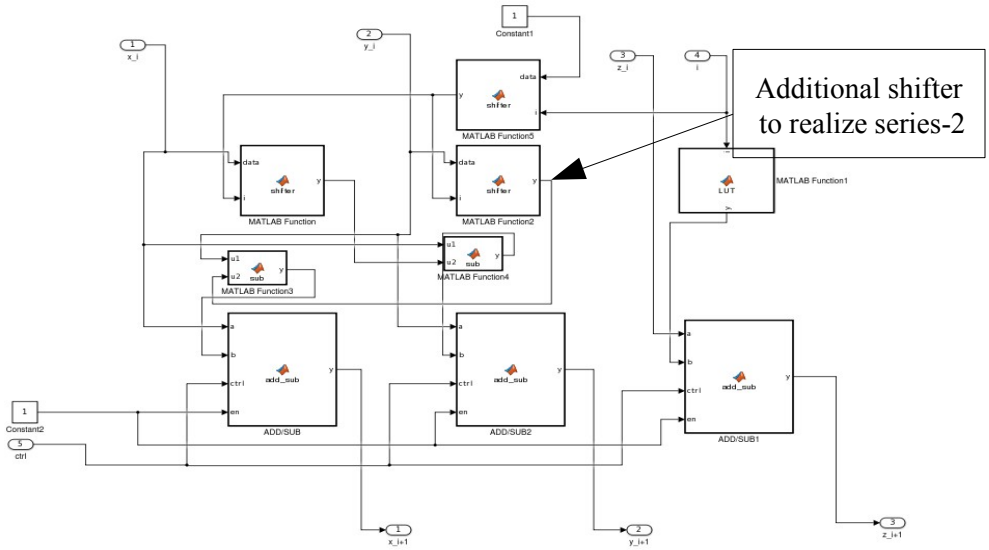

Figure 4.10 Hardware required to implement one stage for i≤0 using series-2

4.17.4 Hardware requirement to implement the series for positive index (for i>0)

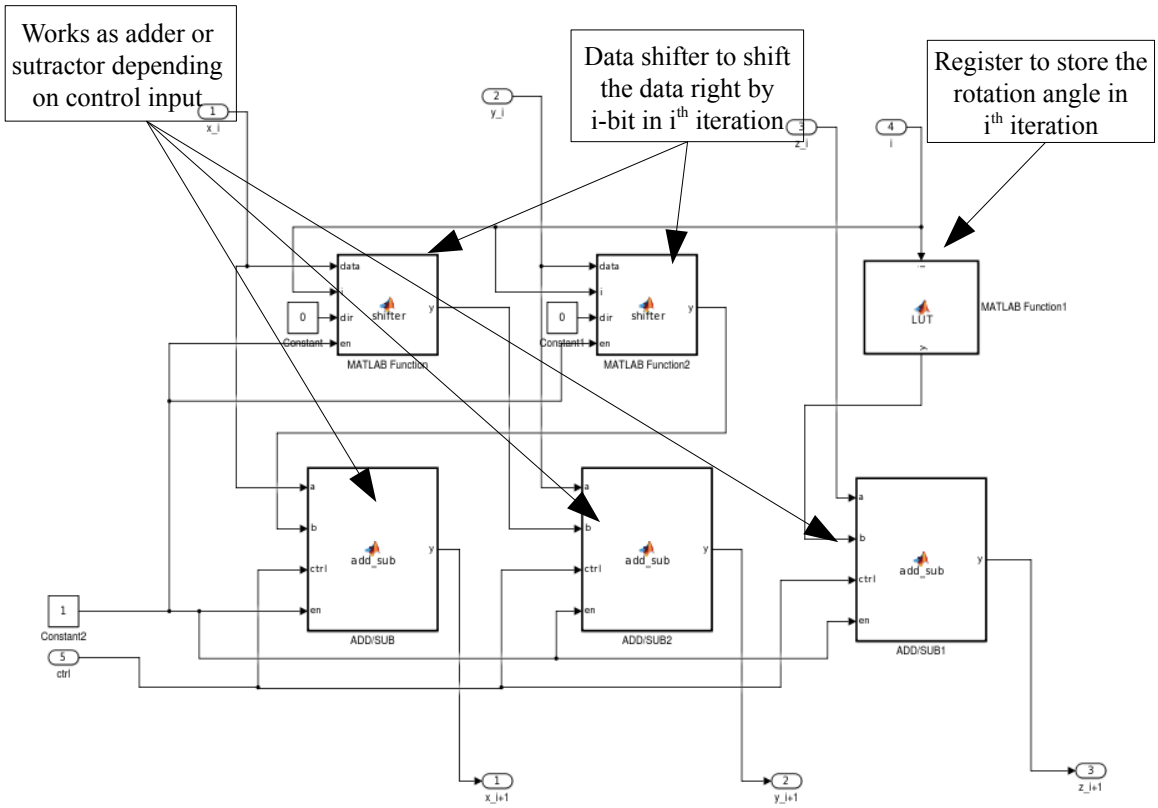

Figure 4.11 Hardware required to implement one stage for i>0 for both the series

4.18 Hardware requirement for different available architectures

There are two different architectures

1. Rotation based architecture
2. Pipelined architecture

In rotation base architecture we need one set of hardware shown in Figure 4.11 (for i>0), one set of hardware either the hardware shown in Figure 4.9 (if we are using series-1) or the hardware shown in Figure 4.10 (if we are using series-2), one LUT that can store all the fixed angles and one control logic block that can iterate the input data for each stage.

Pipelined architecture requires one complete stage for each iteration and here hardware requirement increases with number of iterations.

Figure 4.12 is showing pipelined architecture for logarithmic implementation using series-1. Table 4.12 and Table 4.13 is showing the hardware comparison for series-1 and series-2.

	For positive index			
	total no. of stages	**adder/subtractor**	**shifter**	**register**
Series-1	16	48	32	14
Series-2	16	48	32	14
	For negative index			
	total no. of stages	**adder/subtractor**	**shifter**	**register**
Series-1	7	35	14	7
Series-2	5	25	15	5

Table 4.12 Hardware comparison for series-1 and series-2 for $i>0$ and for $i\leq 0$

Total Hardware Requirement			
	adder/subtractor	**shifter**	**register**
Series-1	83	46	21
Series-2	73	47	19

Table 4.13 Total Hardware requirement

Suppose we need an accuracy of 2^{-14} so we have to go for 14 iterations and in order to ensure the convergence we need to repeat 3rd and 13th iterations and to increase the range so that we can apply a minimum input of 10^{-13}, we need to add 7 more stages for series-1 and 5 more stages for series-2.

Same hardware will work for square-root function also (by doing same manipulation of input data as x=p+1 and y=p-1) but now we need to multiply x' (x co-ordinate of resultant vector) by K/2 to get the square root of p.

4.19 Hardware requirement for different available architectures

As we can see in Table 4.10 and 11 that in comparison to series-2, in series-1, the value of K is changing rapidly and if we are starting from i = - 4 then it is approximately $5*10^{-8}$. So in order to store this result we need a 29 bit register and also we need a multiplier having input data width of 29 bit. If we are computing logarithmic function then there is no need of multiplication with K, since we are dealing with angle. So for logarithmic function series-2 is better than series-1, since lesser stages are required and also hardware requirement is lesser than series-2.

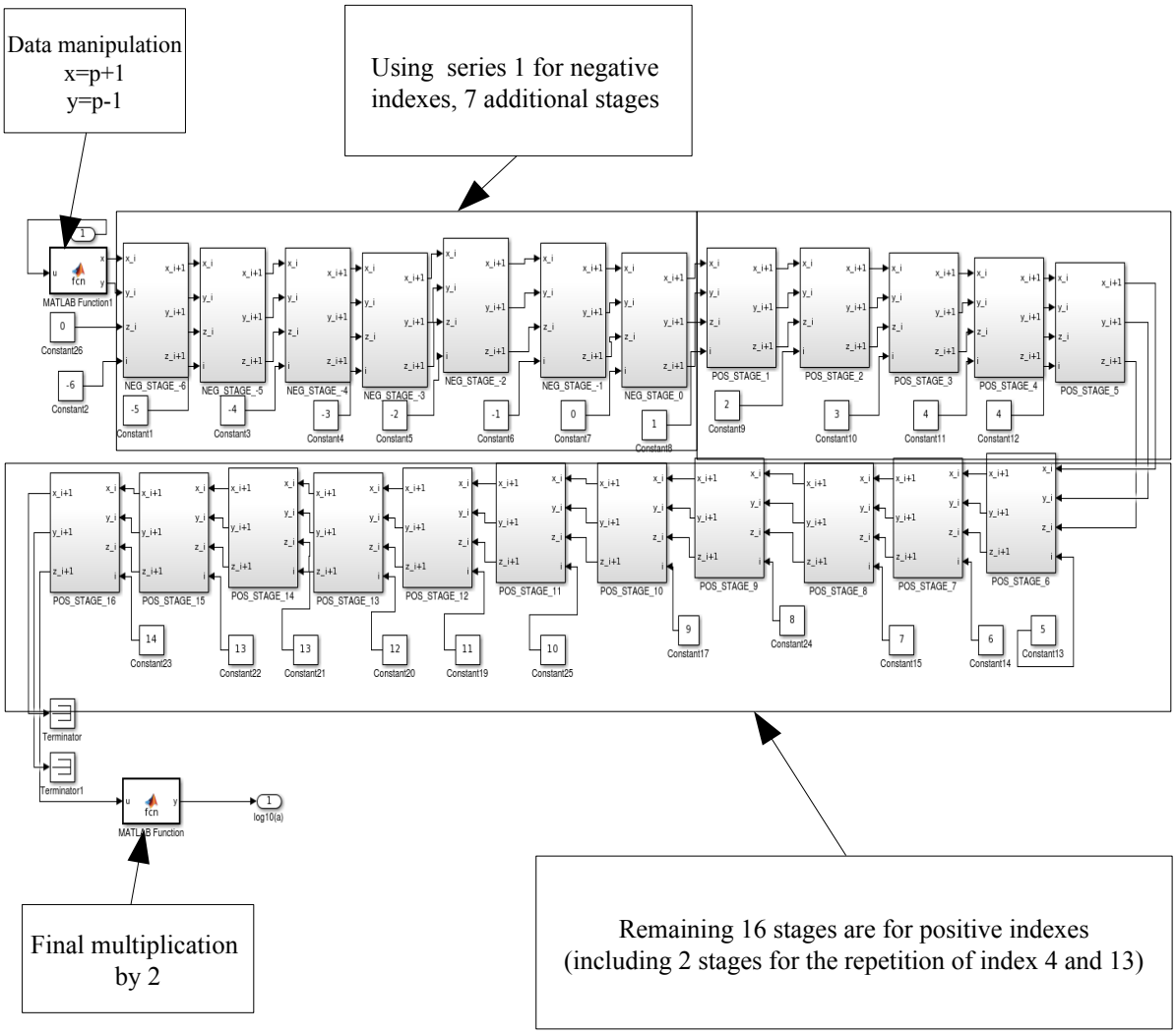

Figure 4.12 Pipelined architecture of series-1 for logarithmic implementation

For series-1 value of K is going up to 22522 (for i starting from -6) and to store this value we need a 16 bit register and multiplier having input data width of 16 bit. Except this issue, series-2 needs lesser hardware (as we can see in Table 4.13).

4.20 Summary

Finally we will implement radix-4 algorithm (decimation in time) for the FFT Engine that requires three complex multipliers. For memory based architecture we need only one radix-4 butterfly. For each radix-4 butterfly, we need 3 complex multipliers (each multiplier will be implemented by CORDIC algorithm). So, the hardware requirement will be 162 adders / subtractors, 102 shifters, 54 comparators (sgn function) and 18 registers to store the values of $\tan^{-1}2^{-i}$.

For the implementation of logarithmic function and square-root function we can utilize the same hardware (for the same manipulation of input data). We just need one additional multiplier and one register (to store the value of K) to implement square-root function. We will use series-2 for the negative indexes since it needs lesser hardware. For an accuracy of 2^{-14}, we have to go for 14 iterations and in order to ensure the convergence we need to repeat 3rd and 13th iterations and to increase the range so that we can apply a minimum input of 10^{-13}, we need to add 5 more stages of series-2. It requires 73 adders/subtractors, 47 shifters, and 19 register for the implementation of logarithmic function and one multiplier and one register for the implementation of square-root function.

Chapter-5

Modeling of Fixed point FFT Engine

5.1 Fixed point modeling and MATLAB constructors

Next step of design flow is the conversion of FFT Engine model from Floating point to Fixed point. At this step of modeling, data width of each hardware is decided according to the need of accuracy. Using a Fixed point Simulink model, one can generate ASIC/FPGA synthesizable RTL code of the model in either VHDL or Verilog using HDL coder (Mathworks) tool. The tool also generates the High-level resource utilization report through that required hardware can be estimated. There are three main constructs in MATLAB to make a fixed point model in MATLAB

1. numerictype (Signedness, WordLength, FractionLength)
2. Fimath (…,PropertyName, PropertyValue,...)
3. fi (data, numerictype, [fimath properties])

5.1.1 numerictype constructor

- Defines signedness, word length and fraction length of data
- The argument **'signedness'** will be **'0'** for unsigned data and **'1'** for signed data

Example:

nt=numerictype(1, 16, 10) will create a signed object with 16 bit word length and 10 bit fraction length

5.1.2 fimath constructor

- fimath(...'PropertyName', 'PropertyValue'...) allows you to set the attributes of a fimath object using property name/property value pairs
- All property names that you do not specify in the constructor, will take the default values of those properties

- fimath object defines arithmetic properties associated with data
- fimath object can also be created using MTLAB Editor.

Example:

```
Fm=fimath('RoundingMethod', 'Floor', …
          'OverflowAction', 'Wrap', ...
          'ProductMode', 'FullPrecision', ...
          'SumMode', 'SpecifyPrecision', ...
          'SumWordLength', 15, ...
          'SumFractionLength', 12, ...
          'CastBeforeSum', true);
```

Above code will create a fimath object given property and values.

5.1.3 *fi* constructor

- 'fi' constructor creates a fixed point data with given numeric type
- One can give a fimath object as an argument to fix the fimath properties of operation

Example:

pi_f = fi(pi,0,8,5) will create an fixed point unsigned variable pi_f with five fractional and three integer bits

5.2 Fixed point modeling of FFT Engine

In the previous chapter, Floating point FFT engine has been designed that takes the real number as input, and processes the data with hardware that can accepts the real input. For the algorithm verification and modeling, floating point model works fine but for the RTL code generation using HDL Coder, modeling must be done into Fixed point where input and output word length of each operation must be defined. As discussed in previous section 'numerictype' construct in MATLAB is used to create an object with given signedness, word length and fraction length and 'fimath' construct is used to create an object with given fimath properties. A Fixed point model in MATLAB behaves as a RTL model in

digital design.

Next step of design flow is the conversion of Floating point model into Fixed point model. In Floating point model, array of persistent variable was used to store the data in FFT block and Analysis block and both the blocks have been designed separately (FFT block and Analysis block are two separate design). In fixed point modeling, both FFT block and Analysis block can be integrated into a single model so that we can generate the RTL code from the top level with appropriate test-bench.

In order to reduce the memory requirement, one septate memory block can be designed that can be allocated to each FFT block as per requirement. Now one can remove the array of persistent variable from both the FFT block and Analysis block. Since the FFT engine is being designed for development of Audio Driver IC. So there might be other blocks (in DSP block) that need memory after the computation of FFT.

To integrate all the three blocks, and to make proper memory allocation algorithm, some status signals must be added to each block that will indicate the start and end of processing in that block. Starting of the process will allocate the memory to that block and end of the process will release the memory access.

5.3 Insertion of memory block

Memory block consist of one FSM, one memory control block, six multiplexers and two RAMs. Multiplexers multiplex the data lines, address lines and control lines of FFT block, Analysis block and other blocks. Properties of Memory block can be summaries as:

- Memory block consist of six multiplexers, two 65536*32 bit RAM, one FSM and one conditional block (to generate control signal for memory)
- Memory block also acts as sampling block that samples the data (2^16 points) from source
- FSM inside the memory block provides the bit reversed address (index) where the sampled data get stored

- 'sampling_start' signal acts as enable signal (active high) for sampling and it is the main control signal in top level that controls the overall flow
- 'sampling_start' signal should be high until the analysis get completed
- 'sampling_start' signal is low means both the RAMs are being used by some other blocks (RAMs can not be used for Sampling, FFT or Analysis)
- To capture another set of sample, make the 'sampling_start' signal low for at least one clock cycle

5.3.1 Memory block in Simulink

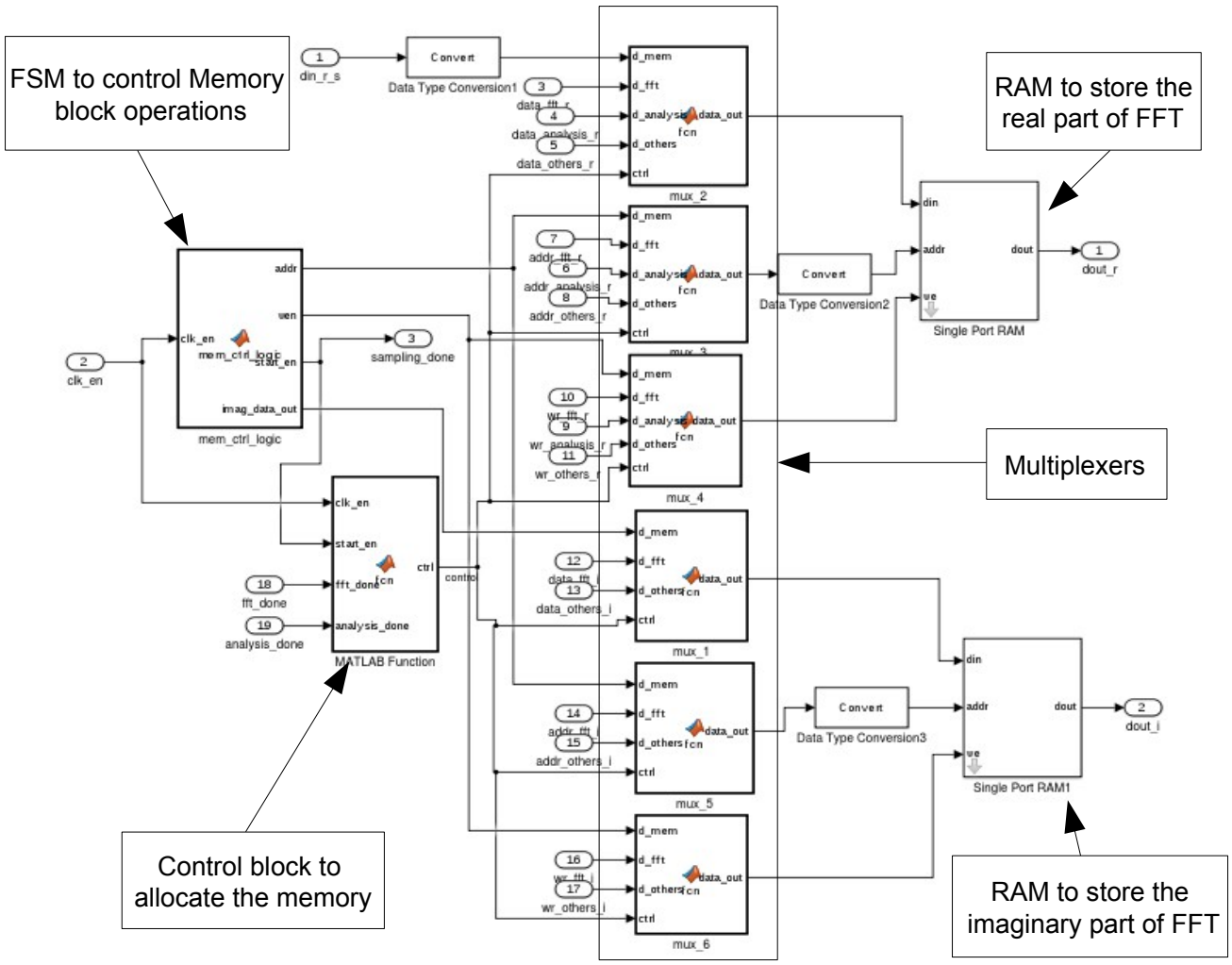

Figure 5.1 Implementation of Memory block in Simulink

Figure 5.1 shows the implementation of memory block in Simulink. To start the sampling of data, 'sampling_start' signal is made high. FSM inside the memory block generates the bit plane reversed index for every received sample. It has been designed to receive 65536 samples and after receiving 65536 samples it makes the 'sampling_done' signal high that indicates that all the 65536 samples have been stored into RAM (that stores the real part of data) at their bit reversed indexes (Index is same as address of RAM). It also clears the Imaginary RAM (RAM used to store the imaginary part of input data) for the same bit reversed address. So 'sampling_start' is the input control pin and 'sampling_done' is output pin where control block sends the status signal.

5.3.2 Pin details of Memory block

Pin Name	Pin Type	Pin Connection	Pin Description	word width
din_r_s	input	external	Input data pin (for sin wave input)	**16**
clk_en	input	external	control signal to start the sampling	**1**
data_fft_r	input	FFT block	Data from FFT block that will be stored in real RAM	**32**
data_analysis_r	input	Analysis block	Data from Analysis block that will be stored in real RAM	**32**
data_others_r	input	external	Data from other blocks (outside the top level) that will be stored in real RAM	**32**
addr_fft_r	input	FFT block	Address of data to be fetched (or to be stored) from (in) real RAM (From FFT block)	**24**
addr_analysis_r	input	Analysis block	Address of data to be fetched (or to be stored) from (in) real RAM (From Analysis block)	**24**
addr_others_r	input	external	Address of data to be fetched (or to be stored) from (in) real RAM (From other blocks)	**24**
wr_fft_r	input	FFT block	Write signal for real RAM from FFT block	**1**
wr_analysis_r	input	Analysis block	Write signal for real RAM from Analysis Block	**1**
wr_others_r	input	external	Write signal for real RAM from other Blocks(outside the top level)	**1**
data_fft_i	input	FFT block	Data from FFT block that will be stored in imaginary RAM	**32**
data_others_i	input	external	Data from other blocks (outside the top level) that will be stored in imaginary RAM	**32**
addr_fft_i	input	FFT block	Address of data to be fetched (or to be stored) from (in) imaginary RAM (From FFT blocks)	**24**

Table 5.1 Pin details of Memory block part-1

Table 5.1 and Table 5.2 show the pin details of the memory block. There are 19 input pins and 3 output pins in the memory block. 'data_others_r', 'addr_others_r', 'wr_others_r', 'data_others_i', 'addr_others_i' and 'wr_others_i' are the data, address and control lines for real and imaginary RAM coming from other blocks (as DSP block).

Pin Naame	Pin Type	Pin Connection	Pin Description	word width
addrr_others_i	input	external	Address of data to be fetched (or to be stored) from (in) imaginary RAM (From other blocks)	**24**
wr_fft_i	input	FFT block	Write signal for imaginary RAM from FFT block	**1**
wr_others_i	input	external	Write signal for imaginary RAM from other Blocks(outside the top level)	**1**
fft_done	input	FFT block	Signal from FFT block that indicates the status of FFT block	**1**
analysis_done	input	Analysis block	Signal from Analysis block that indicates the status of Analysis block	**1**
dout_r	output	I) FFT block ii) Analysis block iii) external	output port of real RAM	**32**
dout_i	output	i) FFT block ii) external	output port of imaginary RAM	**32**
sampling_done	output	i) FFT block ii) external	signal that indicates the status of sampling	**1**

Table 5.2 Pin details of memory block part-2

5.3.3 Single port RAM used in memory block

Figure 5.2 shows the inbuilt single port RAM block used in Memory block. In Single Port RAM we can not read and write simultaneously (in the same clock). In each clock either we can perform either a read operation or a write operation. In Simulink model of Single Port RAM, there is a latency of one clock in reading the data from memory. Two Single Port RAM are being used in the model with size 65536*32 bit each.

One can instantiate the Single Port RAM from 'HDL operations' section of 'HDL coder' library.

We need to select the number of address lines and it will automatically create 2n word space for 'n' bit address line.

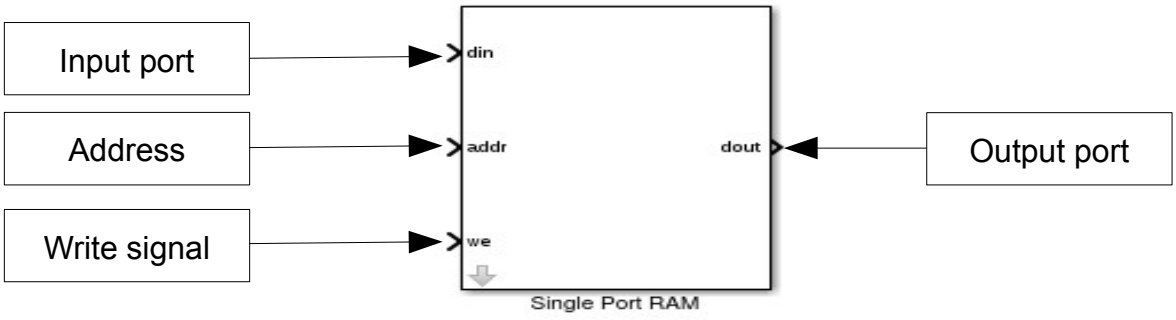

Figure 5.2 Single port RAM model in Simulink

5.3.4 RAM allocation in Memory block

Memory allocation is being done on the basis of two control bits generated by control block. These two control bits act as control signal for all the 6 multiplexers. These multiplexers allow to pass the data line, address line and control line of a particular block depending on control signal. Figure 5.3 shows the memory control block that generates proper control signal for memory allocation. It takes the status signal from all the blocks as input to generate the memory control bits. Table 5.3 shows the allocation of memory depending on control bit status. When both the bits are set then RAMs are allocated to other blocks (Blocks outside the FFT engine).

Memory control bits	Memory Allocation		Process	Signal that indicates the end of process
00	FFT Engine	Memory Block	Sampling	Sampling_done signal goes high
01		FFT Block	FFT computation	fft_done signal goes high
10		Analysis Block	Analysis	Analysis_done goes high
11	Other blocks		-	-

Table 5.3 RAM allocation depending on memory control bits

5.3.5 Parameterization of Memory block

Memory block has been designed for 65536 points and it is not parametrized. To design the Memory block for 1024 points we need to make following changes:

1. Reduce the size of each RAM from 65536 word to 1024 word
2. Change the code for bit plane reversed logic
3. Change the value of Internal parameter N in FSM and make it 1024
4. Change the numeric type of both Data Type Conversion Blocks that go to the address port of RAMs

All the address Registers and Variables have data width of 24 bit and the minimum requirement was 17 bit, so in future, if we need to increase the number of FFT points up to 2^{23}, we need not to change address registers and variables. By making same changes we can also increase the number of Sampling points up to 2^{23}.

5.4 Fixed point modeling of FFT block

Since a separate memory block has been inserted at top level so first we need to modify the original Floating point model, such that it can fetch the data from memory block and restore it after processing. In order to fetch the data from memory block, it has to send proper address and control signal. In previous model, data was stored in an array of persistent variable, so there was no need of address and control line to access the memory. These pins must be inserted before the conversion of Floating point model into Fixed point model. Fixed point model have following properties:

- FFT algorithm is same as used in Floating point model
- Internal structure is for 32 bit (each data has been stored and processed using 32 bit registers and variables)
- All the Registers used for indexes are 24 bit long (n, k, L, N, …)
- In fixed point model, number of stages in CORDIC Multipliers have been increased from 14 to

18

- Hence Fixed point model is more accurate than Floating point model
- In CORDIC block (in CORDIC multiplier as well as logarithmic block used in Analysis block), we have used LUTs to store the angle at each stage
- All the adders and subtractors are for 32 * 32 bit and finally output is getting truncated into 32 bit
- All the angle multipliers are for 24 *24 bit finally output is getting truncated into 32 bits
- In Fixed point model, FFT control logic is fetching the data from memory for processing and after processing it again stores the data back to the memory
- As sampling_done signal goes high, memory is getting allocated to FFT Block
- After computing the absolute value of FFT, 'fft_done' signal goes high and it triggers the processing of Analysis block

'Sampling_done' signal indicates the end of sampling and it starts the FFT computation process. 'fft_done' signal remains low till the computation of FFT and absolute value of computed complex number. As computation ends, it makes the 'fft_done' signal high and memory control block allocates the memory to analysis block.

5.4.1 Pin detail of FFT block

Table 5.4 shows the pin details of FFT block. There are 3 input pins and 7 output pins in the FFT block.

5.4.2 Parameterization of FFT block

- In FFT Block, FSM (control_logic_FFT) is completely parametrized
- 'p' is the parameter that decides the number of FFT points
- Since FFT algorithm is radix-4, so 'p' should be in power of 4 (Ex. 4^n where n is integer)

- In CORDIC multiplier we need to change the value of (2*pi/N) in all the three angel multipliers (Angle_multiplier_1, Angle_multiplier_2, Angle_multiplier_3), as we change the number of FFT points (N)
- To change the clock frequency of FFT block, We just need to change the Sampling Time of FFT block

Pin Name	Pin Type	Pin Connection	Pin Description	word width
memory_data_out_r	input	Memory Block	Data fetched from real RAM	**32**
memory_data_out_i	input	Memory Block	Data fetched from imaginary RAM	**32**
sampling_done	input	Memory Block	signal that indicates the status of sampling	**1**
memory_data_in_r	output	Memory Block	Data to be stored real RAM	**32**
memory_addr_r	output	Memory Block	Address of data to be fetched (or to be stored) from (in) real RAM	**24**
memory_wr_r	output	Memory Block	Write signal for real RAM	**1**
memory_data_in_i	output	Memory Block	Data to be stored in imaginary RAM	**32**
memory_addr_i	output	Memory Block	Address of data to be fetched (or to be stored) from (in) imaginary RAM	**24**
memory_wr_i	output	Memory Block	Write signal for imaginary RAM	**1**
fft_done	output	I) Memory block ii) Analysis block iii) external	Signal that indicates the status of FFT block	**1**

Table 5.4 Pin details of FFT block

5.5 Fixed point modeling of Analysis block

Due to insertion of Memory block at top level, Floating point model model needs to be modified such that it can take the required data from memory and can store intermediate data. There are some other changes in fixed point model in comparison to floating point model and it will be discussed in next section. Properties of fixed point Analysis block are as follows:

- Algorithm is same as used in Floating point model

- All the Registers used for indexes are 24 bit long (registers for max_bin, upper range, lower range, harmonic bin etc)
- Data fetched from memory are stored in 32 bit registers but after squaring its magnitude results are getting stored in 64 bit registers
- There are two multipliers, one adder, one subtractor, one divider and logarithmic block
- Analysis is using only real RAM from Memory block
- As 'fft_done' signal goes high, memory gets allocated to Analysis block
- After computing all the parameters, 'analysis_done' signal goes high and it triggers the allocation of memory to other blocks

'fft_done' signal triggers the computation of desired parameters and allocates the memory to analysis block and 'analysis_done' signal indicates the status of analysis. At the end of analysis, 'analysis_done' signal goes high and memory gets allocated to other blocks.

5.5.1 Changes in fixed point Analysis block

Some changes have been made in Fixed point modeling of Analysis block. These changes have been made to optimize the model in terms of required hardware.

1. **Harmonic bin generation block**

It has been implemented using adders, subtractors and multipliers (There is no dedicated hardware block for harmonic bin generation as in Floating point model). Just one additional 48*24 bit multiplier is required to implement the block. Harmonic bin generation block has been discussed in chapter-2. Now harmonic bin generation takes six clock in Fixed point model where in floating point model, due to dedicated hardware, it takes only one clock cycle to compute the harmonic bin corresponding to given frequency. There is a saving of two multipliers and two dividers in Fixed point model.

2. **A part of code that was computing the 'factor' for each harmonic, has been removed**

It is difficult to set a proper value of 'f_base' since there is a large variation in in the computed values of 'factor'. Now the FFT bin at harmonic position (with proper range) will be assumed as harmonic and it will contribute to total harmonic power.

3. **Percent THD plus Noise computation has been removed**

Since SINAD and Percent THD plus Noise provides the same information but in different way and the computation of Percent THD plus Noise needs one high precision divider and square-root block. Some fixed point computation issues also comes during implementation.

Square root functionality has also been removed from log_sqrt block since it was required for the computation of Percent THD plus Noise and it has been removed, so there is no need of square-root functionality also.

5.5.2 Modeling of Analysis block for Audio and ADC application

Final aim of the project is to design the Analysis block for Audio application which frequency range is 20 Hz to 20 kHz. Floating point model has been designed for ADC application that considers the entire first Nyquist zone (zero to half of the sampling frequency) for the computation of desired parameters. This model can be used to characterize the dynamic performance of an ADC. Same model can be slightly modified for Audio range. Finally there will be two different models, one consisting of Analysis block for Audio range and other will consist of Analysis block that considers entire first Nyquist zone.

1. **Analysis Block for ADC application**

   - For ADC application, frequency range is from zero to Fs/2
   - So our conventional Analysis block will work for ADC application
   - User provides the sampling frequency (Fs) and Input signal frequency (Fin) as input at top-level

- Both the Inputs are in kHz (Ex. If signal frequency is 10 kHz then input should be 10)

2. **Analysis block for Audio application**

- For Audio application, frequency range is from 20 Hz to 20kHz
- Analysis block for Audio application has been modified for the Audio range
- User can select a set of input signal frequency and sampling frequency using three control bits (There are no input ports to feed the input signal frequency and sampling frequency at top level)

5.5.3 Selection of Sampling frequency and input signal frequency for Audio model

Analysis block for audio application has been designed for 8 different set of sampling frequencies and input signal frequencies. Desired set of input signal frequency and sampling frequency can be selected using three control bits. In order to reduce the spectral leakage, coherent sampling has been used where sampling frequency (Fs), input signal frequency (Fm) and number of FFT points (N) satisfies the criteria given in equation (1).

$$k = \frac{N * Fin}{Fs} \tag{1}$$

In equation (1), k is an odd prime integer. Since N is fixed, so we have to chose only those set of Fs and Fm that satisfies the criteria written in equation (1). For ADC model there are two input ports for Fs and Fin, so one can apply any desired set of Fs and Fin that satisfies the coherent sampling criteria but for Audio model, sampling frequency is also fixed (Since it will sample the signal from Audio driver block that has already been designed and it works for 4 different sampling frequencies). Table 5.5 shows the control bit pattern, set of sampling frequencies and input signal frequencies that can be select using a particular pattern. 'fre_sel' input pin Audio model is used to apply the control bit pattern. Input signal near to 1 kHz is industry standard for Audio signal.

Select Bits			Fs (Khz)	Fin (Khz)	FFT Point	Prime No.
SR2	SR1	SR0				
0	0	0	4096	1.0625	65536	17
0	0	1	6553.6	1.1	65536	11
0	1	0	4096	1.0625	65536	17
0	1	1	4096	1.0625	65536	17
1	0	0	5644.8	1.11972	65536	12.999924
1	0	1	6144	1.03125	65536	11
1	1	0	6144	1.03125	65536	11
1	1	1	6144	1.03125	65536	11

Table 5.5 Selection of Fs and Fin for Audio model

5.5.4 Modified Analysis block in Simulink

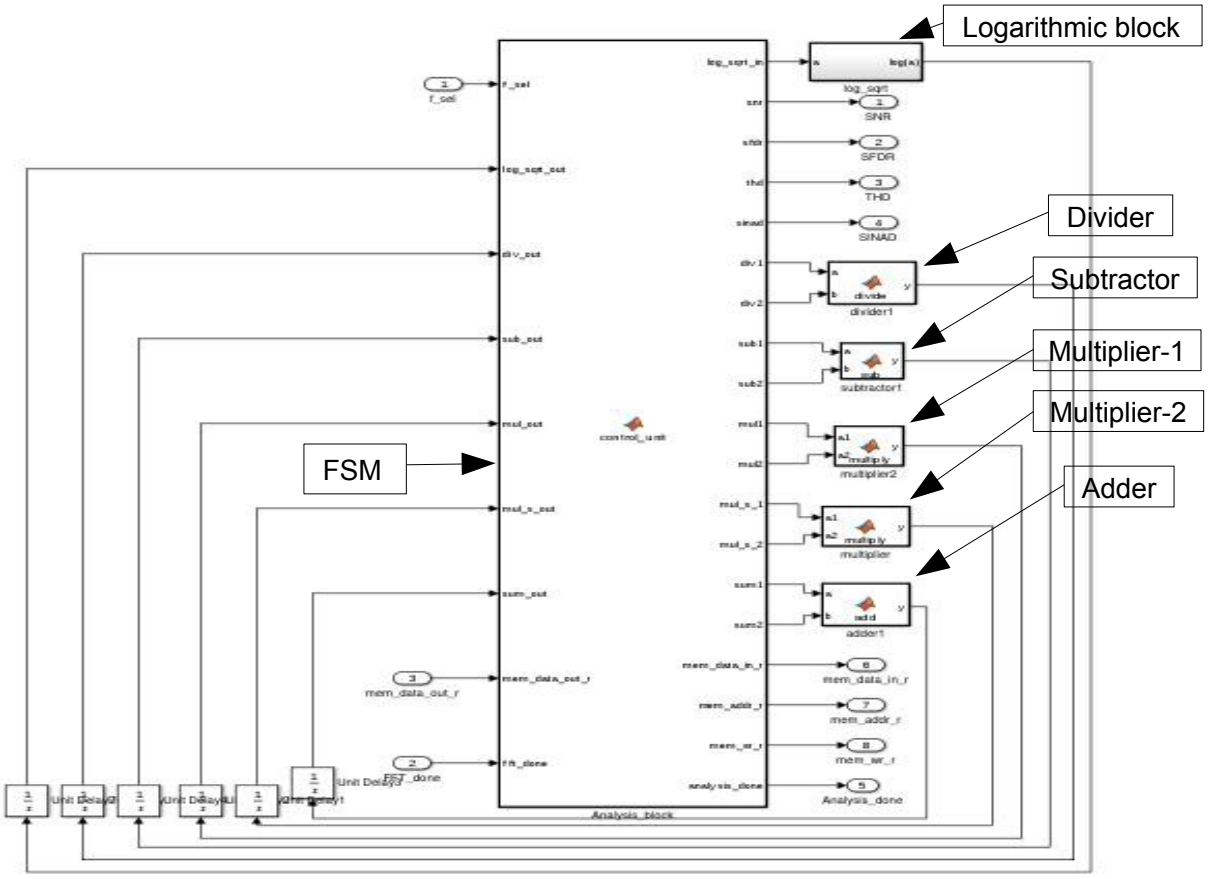

Figure 5.3 Modified Analysis block in Simulink

Table 5.5 shows the control bit pattern, set of sampling frequencies and input signal frequencies that can be select using a particular pattern. 'fre_sel' input pin Audio model is used to apply the control bit pattern. Input signal near to 1 kHz is industry standard for Audio signal.

User should select the proper control bit according to applied input signal frequency and sampling frequency for Audio model.Figure 5.3 shows the modified Analysis block in Simulink. In comparison to Floating point model, it does not have harmonic bin generation block and it has one additional multiplier.

5.5.5 Pin details of Analysis block

Table 5.6 shows the pin details of Analysis block. It has 4 input pins and 8 output pins.

5.5.6 Parameterization of Analysis block

Analysis block is completely parametrized for number of FFT points as well as number of harmonics. 'N' is the parameter to change the number of FFT points (default value is 65536) 'num_harm' is the parameter to change the number of harmonic (default value is 9).

Pin Name	Pin Type	Pin Connection	Pin Description	word width
Fm	input	external	Input signal frequency (only ADC Model)	**32**
Fs	input	external	Sampling frequency (only ADC Model)	**32**
fft_done	input	FFT block	Signal that indicates the status of FFT block	**1**
memory_data_out_r	input	Memory Block	Data fetched from real RAM	**32**
SNR	output	external	computed SNR	**16**
SFDR	output	external	computed SFDR	**16**
THD	output	external	computed THD	**16**
SINAD	output	external	Computed SINAD	**16**
Analysis done	output	I) Memory block ii) external	Signal that indicates the status of Analysis block	**1**
memory_data_in_r	output	Memory Block	Data to be stored real RAM	**32**
memory_addr_r	output	Memory Block	Address of data to be fetched (or to be stored) from (in) real RAM	**24**
memory_wr_r	output	Memory Block	Write signal for real RAM	**1**

Table 5.6 Pin details of Analysis block

5.6 Top model of FFT Engine

Top model of FFT Engine integrates all the three blocks so that one can directly generate RTL code for the entire top model with proper test-bench. Depending on integration of Analysis block, there are two different models of FFT engines, one is for Audio application and other one is for ADC application.

5.6.1 Top model of FFT engine in Simulink

Figure 5.4 shows the top model of FFT engine in Simulink. Data Type Converter block in the model decides the data width of input signal.

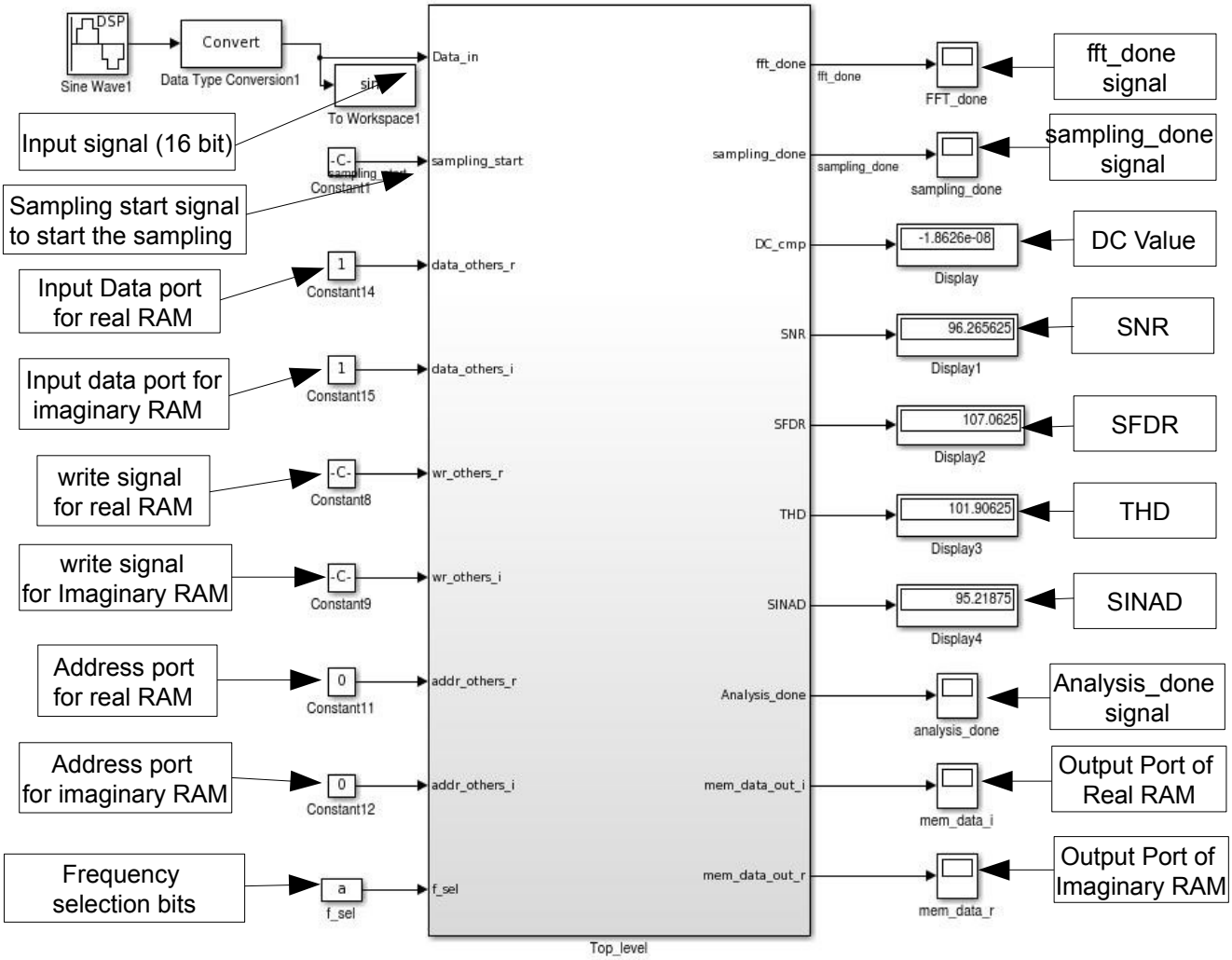

Figure 5.4 Top model of FFT engine for Audio application

We can increase the data width of input signal up to 32 bits since the internal structure has been designed for 32 bits and its pads the zeros before storing it into RAM.

5.6.2 Pin details of FFT Engine

Table 5.7 shows the pin details of FFT Engine for Audio application. There are 9 input pins and 10 output pins in FFT engine. Only the 'fre_sel' pin is not available in ADC model of FFT Engine. Remaining pins are same for both the models.

Pin Name	Pin Type	Pin Connection	word width
din_r_s	input	Input data pin (for sin wave input)	**16**
sampling_start	input	control signal to start the sampling	**1**
data_others_r	input	Data to be stored real RAM	**32**
addr_others_r	input	Address of data to be fetched (or to be stored) from (in) real RAM	**24**
wr_others_r	input	Write signal for real RAM	**1**
data_others_i	input	Data to be stored in imaginary RAM	**32**
addr_others_i	input	Address of data to be fetched (or to be stored) from (in) imaginary RAM	**24**
fre_sel	input	Decides the sampling frequency and Input signal frequency for Analysis block	**3**
wr_others_i	input	Write signal for imaginary RAM	**1**
sampling_done	output	signal that indicates the status of sampling	**1**
fft_done	output	Signal that indicates the status of FFT block	**1**
memory_data_out_r	output	Data fetched from real RAM	**32**
memory_data_out_i	output	Data fetched from imaginary RAM	**32**
SNR	output	computed SNR	**16**
SFDR	output	computed SFDR	**16**
THD	output	computed THD	**16**
SINAD	output	Computed SINAD	**16**
DC Value	output	Computed DC	**32**
Analysis done	output	Signal that indicates the status of Analysis block	**1**

Table 5.7 Pin details of FFT engine for Audio application

5.7 Simulation of FFT engine

FFT engine has been simulated to verify the timing characteristics and computed parameters.

5.7.1 Timing characteristics of FFT Engine

'sampling_start' signal is the only control pin in the Simulink model of FFT engine. In generated RTL code, tool automatically inserts clock enable and reset pins and these things will be discussed in next chapter. Each block has one output pin that shows the status of that block. 'sampling_done' signal shows the status of Memory block, 'fft_done' signal shows the status signal shows the status of FFT block and 'analysis_done' signal shows the status of analysis block.

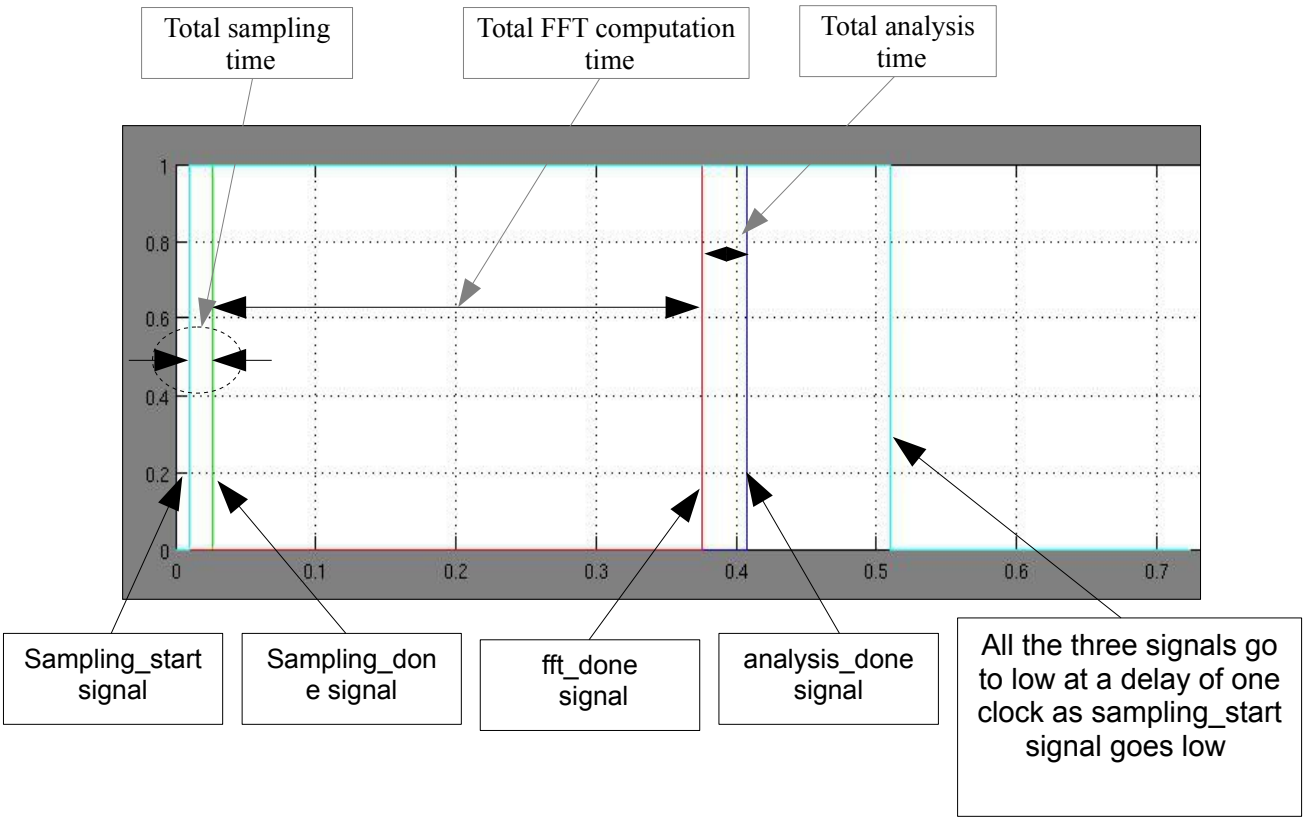

Figure 5.5 Timing characteristics of FFT Engine

Figure 5.5 shows the timing characteristics of FFT engine. Timing diagram is valid for both the

Audio model and ADC model. Once sampling_start signal goes high, we have to keep it high till the completion of analysis (till the analysis_done is low). Once analysis_done goes high, we can make it low at any time.

As 'sampling_done' signal goes low, Memory block makes 'sampling_done' signal low after one clock cycle. By sensing the low level of 'sampling_done' signal, FFT block makes the 'fft_done' signal low after one clock cycle. As 'fft_done' signal goes low, Analysis block makes the 'analysis_done' signal low after one clock cycle.

5.7.2 Verification of ADC model

ADC model can be verified by applying an ideal sine wave quantized with 16 bit ADC and checking the computed SNR against the expected result. For 16 bit ADC expected SNR is approximately 96 dB. For ideal sine wave (no harmonic distortion), SNR will be approximately same as SNDR. Expected THD is less than 100 dB and DC component is less than 1 uV. Ideally expected DC component should be zero and computed total harmonic power should also be zero for ideal sine wave but due to use of CORDIC blocks and truncation of data during processing we can expect some finite error. Computed error should be lees than expected results for ideal sine wave.

Parameters	Computed value
DC	-1.12E-08
SNR	96.14
SFDR	116.75
THD	113.98
SNDR	96.06

Table 5.8 Computed parameter ADC model

Table 5.8 shows the simulation results for ADC model. Commuted results are approximately matching with expected results. Results have also been cross checked using Cadence tool by exporting the input data to Cadence using Verilog-A code.

5.7.3 Verification of Audio model

Since audio model has been designed for the development of Audio Driver IC where the output of DSP block is applied to FFT engine. DSP block consist of a Noise Shaper block that alters the spectral shape of noise. It decreases the noise level for audio band and increases the noise level outside the audio band. Figure 5.6 shows the FFT plot at output port of DSP block. Due to nonlinear nature of noise floor it is difficult to compute the SNDR mathematically. So to verify the FFT engine we have to cross check the computed results against the results computed by Cadence tool.

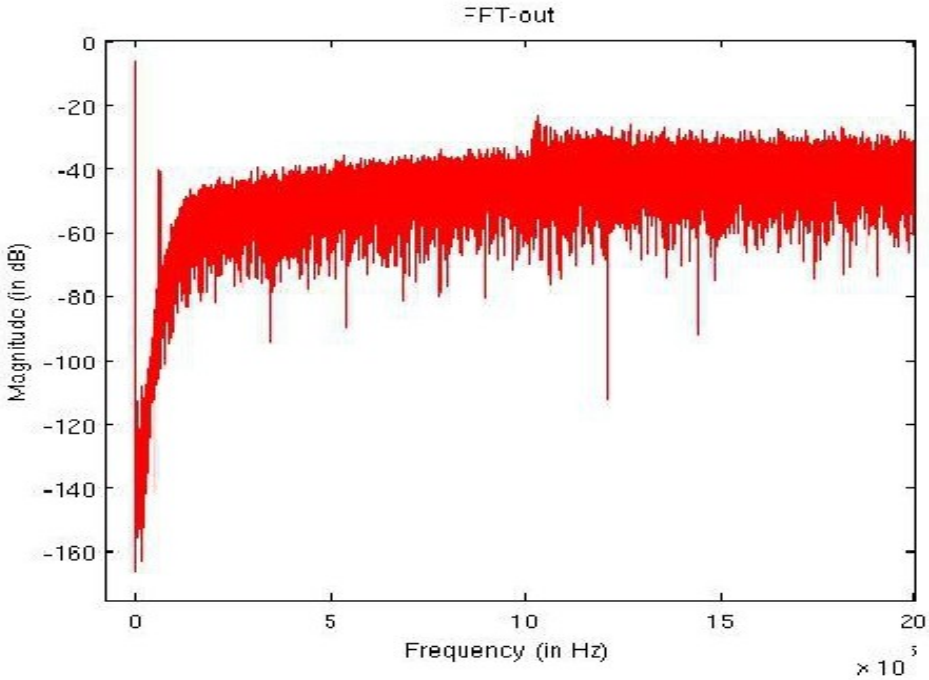

Figure 5.6 FFT plot at Noise Shaper output

Figure 5.7 shows the test-bench for the verification of FFT engine. In this test-bench ideal sine wave is quantized using 11 bit quantizer to increase the quantization noise.For 16 bit quantization, noise floor in audio band is going bellow 130 dB (If computed using Cadence tool) but designed FFT block is not that much accurate to compute this noise floor.

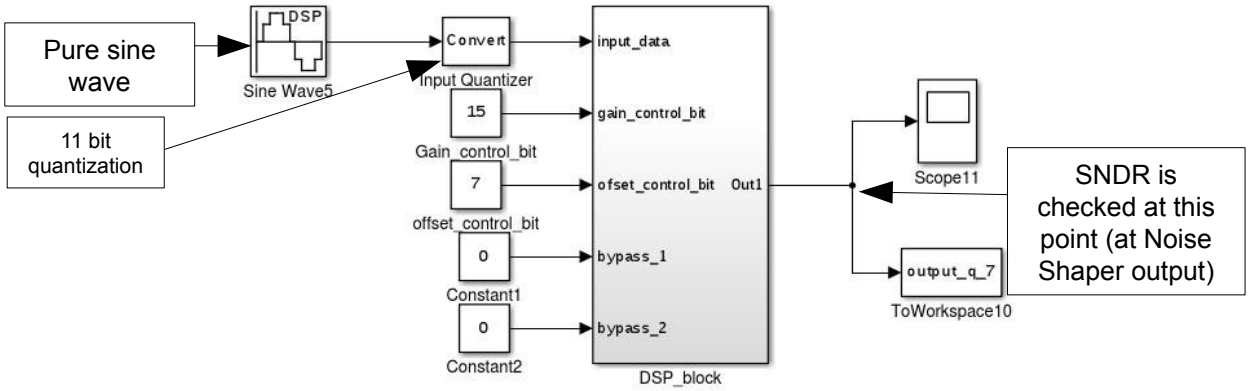

Figure 5.7 Test-bench for Audio model

Computed SNDR using Cadence tool	Computed SNDR using Audio Model
89.20 dB	89.15 dB

Table 5.9 Computed SNDR using FFT engine and Cadence tool

Table 5.9 shows the computed SNDR using FFT engine as well as using Cadence tool. Results are approximately matching with an error of 0.05 dB.

Chapter-7

RTL Code generation and RTL verification of FFT Engine

6.1 RTL code generation

Next step of implementation is the generation of synthesizable RTL code and test-bench for FFT engine using the designed Fixed point model. For each part of MATLAB code inbuilt Simulink blocks, tool generate corresponding HDL code.

6.1.1 HDL Coder tool

HDL Coder is a tool from MATLAB that generates RTL code from Fixed point Simulink model. Tool needs a separate license for HDL Coder along with Simulink license. In this project HDL Coder 3.8 has been used for RTL code generation. Targeted hardware is ASIC (Application Specific Integrated Circuit). Tool can also synthesize the generated code for a specific FPGA (from Xilinx and Altera) but supported software should be installed and its path should be attached with MATLAB. Tool can generate the HDL code in both Verilog and VHDL languages.

Along with RTL code and test bench tool also generates:

1. **Resource Utilization Report:**

Resource utilization report provides a detailed estimate of required hardware (adder, subtractor, multiplier, multiplexer, RAMs and registers).

2. **Traceability Report:**

Traceability report maps each MATLAB code with corresponding HDL code. So we can trace the generate HDL code for each Simulink block as well MATLAB code

3. **Critical path report:**

Asserts the critical path timing in the simulink model. It helps to identify the speed bottelnecks to improve the design performance.

4. **Model web view:**

Creates a web view of complete model.

6.1.2 Input ports inserted by HDL Coder tool

During HDL code generation, tool adds three more input pins in addition to per-defined input ports:

1. **Input port for clock signal (default name 'clk'):**

   This input port is used to provide clock signal to the RTL model.

2. **Input port for asynchronous reset signal (default name 'Reset'):**

   This pin is used as global asynchronous reset.

3. **Input port for clock enable signal (default name 'clk_enable'):**

   This pin is used as clock enable pin for RTL model.

6.1.3 Clock input Pins

We can insert more than one clock input port for multi rate models. We can select either either single clock port or multiple clock port for code generation.

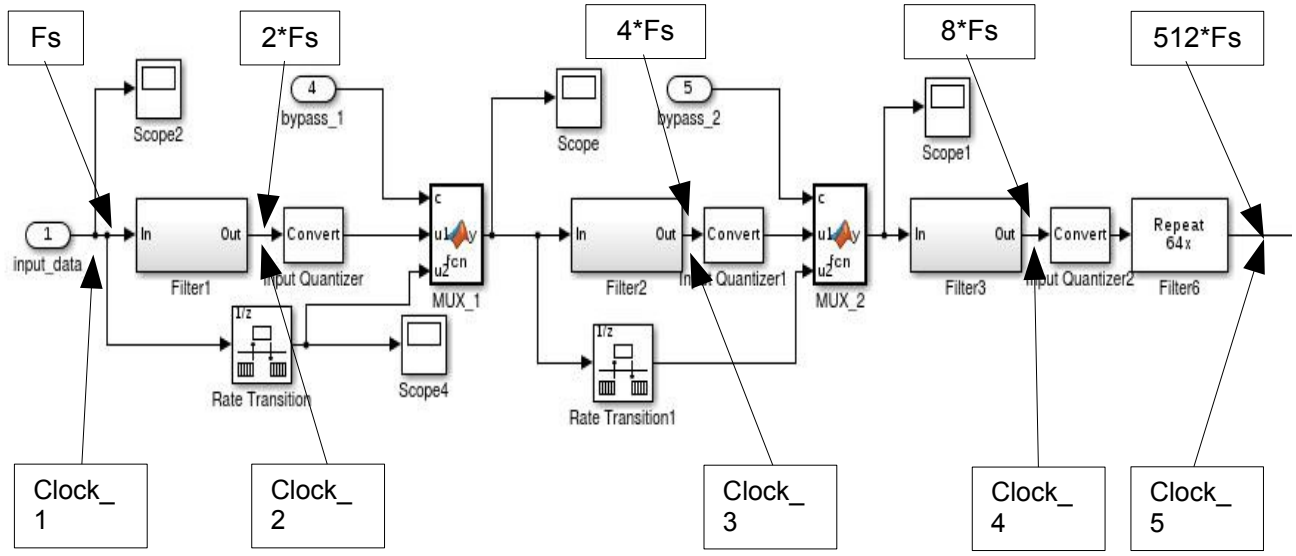

Figure 6.1 Multi-rate Simulink model

Clock input => single: it will generate a single clock input port at top level.

Clock input => multiple: number of clock input port is dependent on model. If blocks are running at different speed then it will generate more than one clock input but if blocks are running at same speed then it will generate single clock input port. Figure 6.1 shows a Simulink consisting of interpolation filters and repeaters. Model is running at 5 different sampling frequencies, so the tool generates five different clock input ports.

6.1.4 Example of Verilog Code generation

Verilog code has been generated for a fixed point Simulink model of adder, created using 'user defined MATLAB function' block.

MATLAB code for 14 bit fixed point adder

```
function y = adder(u1,u2,en)
%#codegen
u1=fi(u1, 1, 14,12);
u2=fi(u2, 1, 14,12);
fimath('RoundingMethod',
'Nearest', ...
   'OverflowAction', 'Wrap', ...
   'ProductMode', 'FullPrecision', ...
   'SumMode',
'SpecifyPrecision', ...
   'SumWordLength', 15, ...
   'SumFractionLength', 12, ...
   'CastBeforeSum', true)
if en==true
 y = fi(u1+u2,1,15,12,fm);
else
   y=fi(0,1,15,12);
end
```

HDL code generation

Generated verilog code

```
module MATLAB_Function (u1, u2, en, y );

  input   signed [13:0] u1;  // sfix14_En12
  input   signed [13:0] u2;  // sfix14_En12
  input   en;
  output  signed [14:0] y;  // sfix15_En12


  reg signed [14:0] y_1;  // sfix15_En12
  reg signed [14:0] add_cast_1;  // sfix15_En12
  reg signed [14:0] add_cast_0_1;  // sfix15_En12


  always @(u1, u2, en) begin
    if (en == 32'sd1) begin
      add_cast_1 = u1;
      add_cast_0_1 = u2;
      y_1 = add_cast_1 + add_cast_0_1;
    end
    else begin
      y_1 = 15'sb000000000000000;
    end
  end
  assign y = y_1;
endmodule
```

Figure 6.2 Verilog code generation example

As shown in Figure 6.2, in the left hand side, MATLAB code for 14 bit fixed point adder is written and generated Verilog code for 14 bit adder is written in another side. One can observe that the tool also generates 'en' input port in addition to adder ports.

6.2 RTL code and test-bench generation for FFT engine

RTL Code and test bench is generated using the designed fixed point model. Top level of FFT engine consist of three more input ports, in addition to ports shown in chapter-4, Table 6.8. Since whole design is working on a global clock signal, so tool is generating a single clock input port.

Tool is also generating a test bench for the verification of generated RTL code. Tool stores each input and output data for every step of simulation into different arrays. Then in test bench code, it feeds all the stored input to DUT and stores the results computed by DUT into another set of arrays. It compares the computed results with stored results at each simulation step. If both the results are matching then it displays the message " Test Completed (Passed)". If at any simulation step, both the results are not matching then it displays the message " Test Completed (Failed)".

6.2.1 Resource utilization report

Tool estimates the required hardware for the generated HDL code. It estimates the number of required multipliers, adders, multiplexers, subtractors, RAMs and registers.

Resources	Required number of resources
Multipliers	16
Adders/Subtractors	689
Registers	87
RAMs	2
Multiplexers	525

Table 6.1 Required resources for FFT engine

Table 6.1 shows the resourced utilized by FFT engine that consists of FFT block, Analysis block and Memory block.

6.2.2 Resource advantage of FFT engine over HDL optimized FFT block

HDL Coder library in Simulink, provides HDL optimized FFT block through that one can directly generate the FFT block. This block uses radix-2 algorithm and pipelined architecture. Verilog code has been generated for HDL optimized FFT block for the same number of points. Resource utilization summary shows that FFT engine requires less hardware in comparison to HDL optimized FFT block. Table 6.2 shows the comparison of required resources for FFT engine and HDL optimized FFT block.

Resources	Using HDL optimized FFT block	FFT block + Memory Block +Analysis Block
Multipliers	28	16
Adders/Subtractors	366	689
Registers	1439	87
RAMs	34	2
Multiplexers	924	525

Table 6.2 Comparison of Required resources for FFT Engine and HDL optimized FFT block

Generated code can be synthesize on ASIC or FPGA. Aim of the project is to synthesize the RTL code on ASIC. Synthesis is the process in which tool take the RTL code (Verilog or VHDL), targeted technology, and constraint as inputs and maps the RTL to target technology primitives. Synthesis tool may be Design Compiler from Synopsis.

6.3 RTL verification using ModelSim

RTL is simulated using ModelSim (from Altera). Generated FFTT test-bench has been used for the RTL verification. Test-bench is generated using the Fixed point Simulink model so the computed

result should be same.

Figure 6.3 Simulation result of RTL code

Figure 6.3 shows the Simulation results FFT engine designed for ADC application. One can observe that, as 'analysis_done' signal goes high, all the computed parameters are reflected at the output ports.

We can compare the result computed RTL and fixed point model and verify the results. For each parameter, hexadecimal radix is chosen. So to check the result we need to convert the hexadecimal result into decimal. Word length for SNR, SFDR, SNDR and THD is 16 bit with 6 fractional bits and 1 sign bit.

Parameters	Results for Fixed point FFT engine	Results for RTL simulation (Hex format)	Results for RTL simulation (Dec format)
DC	-1.12E-08	fffffffa	-1.12E-08
SNR	96.14	1809	96.14
SFDR	116.75	1d30	116.75
THD	113.98	1c7f	113.98
SNDR	96.06	1804	96.06

Table 6.3 Comparison of RTL simulation results and Fixed point simulation results

Table 6.3 shows the results computed by DUT in hexadecimal form, its decimal equivalent value and the results commuted by fixed point FFT Engine designed for ADC application. Both the results are exactly matching. So both the RTL code and test-bench are working fine.

Chapter-7

Conclusion, Scope of improvement and Future work

7.1 Conclusion

FFT Engine has been designed at three different level of abstraction, Algorithmic level, Micro-design or low level design and RTL code generation. FFT engine is divided into three blocks, Memory block, FFT engine and Analysis block. FFT block is commuting FFT using radix-4 algorithm and it has been implemented using Memory based architecture. FFT block will compute 2^{16} (65536) point FFT. CORDIC multipliers are used to save the memory. Analysis block has been designed for audio application that uses audio range (20 Hz to 20 kHz) for analysis as well as for ADC application that uses entire first Nyquist zone (zero to half of the sampling frequency).

FFT engine can compute the parameters SNR, SNDR ,SFDR, THD and DC component for the noise floor up to -120 dB with an error less than 0.5 dB.

There is huge saving of resources utilized by FFT Engine in comparison to HDL optimized FFT block given in Simulink HDL Coder library . HDL optimized FFT block itself uses 28 multipliers, 366 adders/subtractors, 1439 registers, 34 RAMs and 924 multiplexers. Our FFT block along with Memory block and analysis block needs only 16 multipliers, 689 adders/subtractors, 87 registers, 2 RAMs and 525 multiplexers. Number of adders are more because of logarithmic block and CORDIC block that have been designed for 21 stages and 19 stages respectively and each stage needs 6 adders/subtractors.

Generated RTL code (Verilog) has is simulated using ModelSim and results have been verified with the results computed by Cadence tool as well as the result computed by Fixed point FFT Engine.

7.2 Scope of improvement

Scope-1:

In CORDIC block, Adder/Subtractor block has been used where a control bit decides whether this block will act as Adder or Subtracter. HDL Coder is generating one Adder, one Subtractor and one multiplexer to select the appropriate hardware according to control signal. In digital design, one can use

xor gates and adder to perform the operation. We can save huge number of adders using this technique.

Scope-2:

Logarithmic block is operating on maximum data width (64 bit). We can try this block using series-1 which needs 2 more stages but the value of constant 'K' is less for same stages.

Scope-3:

All the registers used for indexes have been taken for 24 bit and minimum requirement is 17 bit (In all the three blocks). We have taken it because everywhere in the model data width is in bytes (multiple of 8 bit).

7.3 Future Work

1. **Synthesis of RTL code:**

Next step of design flow is the synthesis of RTL code with the tool like Design Compiler to get the gate-level netlist for the targeted technology and constraints. The designed FFT engine is an integral part of Audio Driver IC, and it will be integrated for the development of Audio Driver IC.

2. **Place and Route:**

It is the next level of design flow. Place and Route tool takes the gate-level netlist as input and output is a GDS file, used by foundry for fabricating the ASIC.

3. **Design of Control block:**

Computed results can be analyses through the a Control block and can be offset by sending proper feed back signal. As one can take the example of computed DC parameter, if it going beyond a specific quantized level then control block may subtract the that quantized level from each sample that will reduce the net offset in the signal by that quantized level.

References

[1] Xin Xiao, Erdal Oruklu, and Jafar Saniie, IEEE Transactions on circuits and systems-ii: Express Briefs, VOL. 55, NO. 11, November 2008.

[2] Prof. J.M.Rudagi , Richard Lobo, Pradeep Patil, Nikit Biraj, Naimahmed Nesaragi , International Conference on Advances in Recent Technologies in Communication and Computing , 2010.

[3] M. Hasan, T. Arslan and J.S. Thompson , A Novel Coefficient Ordering based Low Power Pipelined Radix-4 FFT Processor for Wireless LAN Applications , IEEE Transactions on Consumer Electronics, Vol. 49, No. 1, February 2003.

[4] Xiaobo Hu, Ronald G. Harber and Steven C. Bass, Expanding the Range of Convergence of the CORDIC Algorithm, IEEE Transactions on Computers, VOL. 40, NO. 1, January 1991.

[5] Texas Instruments, FFT Implementation on the TMS320VC5505, TMS320C5505, and TMS320C5515 DSPs.

Available: www.ti.com/lit/an/sprabb6b/sprabb6b.pdf

[6] John G. Proakis , Dimitris K Manolakis, Digital Signal Processing: Principles, Algorithms, And Applications, Third Edition,Efficient computation of the DFT: Fast Fourier Transform algorithms, Prentice Hall International INC.

[7] Texas Instruments, Implementing the Radix-4 Decimation in Frequency (DIF) Fast Fourier Transform (FFT) Algorithm Using a TMS320C80 DSP.

Available: http://www.ti.com/general/docs/lit/getliterature.tspliteratureNumber=spra152 &fileType=pdf

[8] MathWorks, Getting Started with MATLAB to HDL.

Available: Workflow,http://in.mathworks.com/help/hdlcoder/examples/getting-started-with -matlab-to-hdl-workflow.html

[9] Mathworks, HDL Coder.

Available: www.mathworks.in/products/datasheets/pdf/hdl-coder.pdf

[10] Tzi-Dar Chiueh and Pei-Yun Tsai, Circuit techniques/ FFT Algorithms, OFDM Baseband Receiver Design for Wireless Communications , John Wiley and Sons (Asia) Pvt. Ltd.

[11] MathWorks, signal processing toolbox.

Available: http://in.mathworks.com/help/signal/index.html

[12] MathWorks, Fixed-Point Designer.

Available: http://in.mathworks.com/help/fixedpoint/index.html

[13] Analog Devices, Understand SINAD, ENOB, SNR, THD, THD + N, and SFDR so You Don't Get Lost in the Noise Floor.

Available: http://www.analog.com/media/en/training-seminars/tutorials/MT-003.pdf

[14] Ian Beavers , Understanding Spurious-Free Dynamic Range in Wide-band GSPS ADCs

Available: http://www.analog.com/media/en/technical-documentation/technical-articles/Understanding-Spurious-Free-Dynamic-Range-in-Wideband-GSPS-ADCs-MS-2660.pdf