The Thesis Committee for Muralidharan Jagannathan
Certifies that this is the approved version of the following thesis:

Impedance Measurement Device: A System-on-Chip implementation

APPROVED BY
SUPERVISING COMMITTEE:

Supervisor:

Jacob A. Abraham

Mark W. McDermott

Impedance Measurement Device: A System-on-Chip implementation

by

Muralidharan Jagannathan, BE

Thesis

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

May 2010

Acknowledgements

I would like to acknowledge Prof. Jacob A. Abraham, Prof Arjang Hassibi and Prof. Mark McDermott for providing me with this opportunity to do this thesis work with them. I would like to acknowledge their support and guidance during the execution of this thesis. I would like to thank Arun Manickam, who worked with me during this thesis to design and implement the analog components of the SoC.

This has been a wonderful experience and has helped me to understand the integrated circuit design process better.

05-May-2010

Abstract

Impedance Measurement Device: A System-on-Chip implementation

Muralidharan Jagannathan, MSE

The University of Texas at Austin, 2010

Supervisor: Jacob A. Abraham

This System-on-Chip implementation is aimed at measuring small impedance with high accuracy. The system consists of an Analog Sensor, Analog to Digital convertor and a computational unit for calculating the magnitude and phase of the impedance being measured. This thesis deals with the computational unit that is used to calculate the magnitude and phase of the impedance.

The SoC can find its application in affinity based biosensors that use impedance spectroscopy to determine the properties of the analyte. Other applications include measuring impedances from probes buried in building structures to monitor the health of the buildings.

Table of Contents

List of Tables

List of Figures

CHAPTER 1: INTRODUCTION

Biosensors are devices used for detection of a biological analyte using a biological component and a physicochemical component. Most of the biosensors are affinity-based which takes advantage of the ability of bio-molecular pairs (probe-analyte pair) to favorably bind and form a structure with lower overall energy (1). Affinity based biosensors make use of different methods of detection based on optical (2), electrochemical (3) (4) (5), magnetic (6), surface plasma resonance (7) etc. Among the various methods, electrochemical impedance spectroscopy has generated some interest in the recent years (8). Impedance spectroscopy allows label-free and real-time detection. The use of labels could lead to significant increase in cost and complexity of the bio-molecular detection protocols. Also varying labeling efficiency could lead to measurable variation in the detection process. Also, impedance spectroscopy permits real-time detection, which allows researchers to gather information about reaction kinetics and bio-molecular interactions.

In an affinity based biosensor the probes are immobilized on a planar surface and when the solution containing the analyte is added, due to the affinity between the probe and the analyte, the analyte gets attached to the probes. During the detection step, the number of analyte-bindings is counted in order to estimate the original analyte concentration.

In an impedance spectroscopy based affinity biosensor, the surface on which the probes are fixed is a metal electrode. The probe-analyte binding causes the interfacial impedance to change, leading to the detection of the binding.

We can assume that the analyte would affect both the resistive and reactive components of the impedance. Hence the problem in hand is to measure both the magnitude and phase of the impedance. The impedance of the system is measured at various frequency points, which could be used to determine the changes in the impedance with probe-analyte binding and hence detect the presence of the different analytes in the sample.

Apart from bio-molecular detection, impedance spectroscopy could be used for a wide variety of applications. One potential application is in the area of structural health monitoring, in which the sensor is buried in the steel structures and changes in impedance due to corrosion can be detected remotely by using impedance spectroscopy (9).

CHAPTER 2: THE SYSTEM

The aim of the system is to measure impedance. To achieve this we define a system that can sense impedance and calculate the magnitude and phase of it at various frequencies, thereby enabling impedance spectroscopy.

The system can be split into two subsystems, the analog subsystem and the digital subsystem. The analog subsystem consists of all the analog circuitry of the system and the digital subsystem consists of the backend computational circuitry.

The analog subsystem consists mainly of two units, the sensor (9) and the analog to digital convertor. The sensor circuit senses the impedance and converts it to electrical signals. The sensor outputs two signals, an “in-phase” signal, “I” and a “Quadrature” phase signal “Q” and both these signals are differential signals.

The Sigma-Delta Analog-to-Digital convertor (11) is used to convert these analog outputs of the sensor to digital signals. In order to save power and silicon area, we use one ADC in a multiplexed mode. The ADC first converts the data from the “I” channel and then it is switched to the “Q” channel to do the conversion. This also helps to make sure the offset, if any, out of the ADC to be same for both “I” and “Q” channels.

The digital subsystem cannot directly handle this ADC output. The ADC’s output is an oversampled signal and is a one bit stream. Hence the frontend logic in the digital subsystems decimates the ADC output and is fed to the computational portion of the digital subsystem. The digital subsystem is discussed in detail in the following chapters.

CHAPTER 3: INTRODUCTION TO DIGITAL SUBSYSTEM

The sensor gives a two channel output, the in-phase channel, I, and the Quadrature phase channel, Q. We use one ADC to convert both I and Q channel analog data into 16-bit digital. Now the problem is to find the magnitude and phase in the digital I and Q channel and ship the result out of the system.

The magnitude of a signal represented by I and Q is given by

$$Mag = \sqrt{I^2 + Q^2}$$

The phase of the signal represented by I and Q is given by

$$Phase = \tan^{-1}\left(\frac{Q}{I}\right)$$

These calculations are fairly complex for a digital unit. The first operation (magnitude) has two square operations followed by a square root operation. The second operation (phase) has a division operation followed by a tan inverse operation. One way to implement them to realize the function as a whole in logic and another way would be to go for a unit that can perform these operations easily like a COordinate Rotation DIgital Computer (CORDIC). The first method calls for implementation which uses very slow and complicated logic which might prove to very inefficient and power consuming. The second system is efficient and easy to implement (12).

The other main function of the digital sub-system is to transfer the value computed the CORDIC unit to the outside world; for this purpose we use an I^2C interface. I^2C is a simple two wire interface invented by Philips and is widely used by many IC manufactures. This will allow us to connect this chip to any other I^2C compatible device.

CHAPTER 4: CORDIC

4.1 Introduction

COordinate Rotational DIgital Computer or CORDIC is a simple and efficient algorithm to calculate hyperbolic and trigonometric functions (13). CORDIC performs the complex function using a set of rotations performed by adders, thereby eliminating the need for multipliers. It is an ideal solution where area and power are a main consideration. CORDIC was first described by Jack E. Volder in 1959 (11). Some of the main applications of CORDIC are trigonometric calculations, Modulation, PLL, phase rotation, FFT, DCT, etc (12).

4.2 CORDIC Operation

The basic idea of a CORDIC processor is to take a vector in Cartesian Co-ordinate system and rotate it by a certain fixed angle. Rotating a vector by an arbitrary angle is difficult. Hence to rotate a vector by angle 'θ', we perform a set of rotations. The angle by which each rotation is performed is predetermined and is fixed. The number of rotations determines the accuracy of the system.

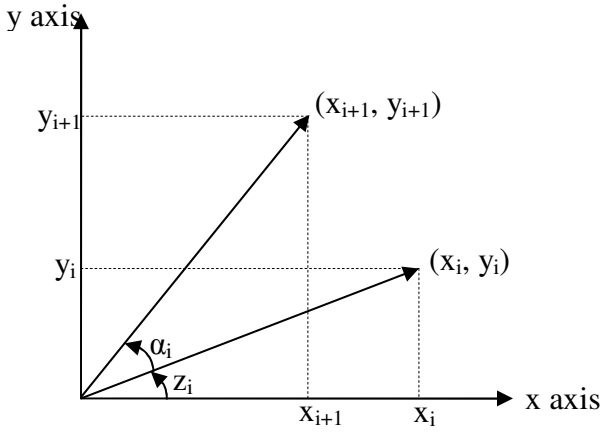

Figure 1: CORDIC Rotation Example

$$\theta = \alpha_1 + \alpha_2 + \alpha_3 + \alpha_4 \ldots . + \alpha_n$$

where θ is the desired angle of rotation and α_1, α_2 to α_n are the angles of the individual pseudo rotation to be performed and n is the number of pseudo rotation.

Suppose we rotate a vector OA with initial end points at $(x_i,\ y_i)$ by α_i. Let us assume the final end point is $\left(x^{i+1}, y^{x+1}\right)$. Let us assume the initial angle is "z" Then the final end point is given by:

$$x_{i+1} = x_i \cos\alpha_i - y_i \sin\alpha_i$$

$$y_{i+1} = y_i \cos\alpha_i + x_i \sin\alpha_i$$

$$z_{i+1} = z_i - \alpha_i$$

These equations can be re-written as:

$$x_{i+1} = \frac{(x_i - y_i \tan\alpha_i)}{\sqrt{1 + (\tan\alpha_i)^2}}$$

$$y_{i+1} = \frac{(y_i + x_i \tan\alpha_i)}{\sqrt{1 + (\tan\alpha_i)^2}}$$

$$z_{i+1} = z_i - \alpha_i$$

These equations still look complex but we need to note that if we choose α^i such that $\tan\alpha^i$ is a power of two then the tan operation simply becomes a digital shift operation which can be performed easily. More over since α is a constant the expression $\sqrt{1 + (\tan\alpha_i)^2}$ is constant for particular rotation iteration. For a set of "n" rotation the contribution due to this factor is a constant. Hence we can remove this expression in a CORDIC rotation and call the resultant as CORDIC pseudo rotation. We will keep track of these values and finally apply a correction factor. The expression for pseudo rotation is:

$$x_{i+1} = (x_i - y_i \tan\alpha_i)$$

$$y_{i+1} = (y_i + x_i \tan\alpha_i)$$

$$z_{i+1} = z_i - \alpha_i$$

The above equation gives the next values of x and y when we are doing a positive rotation. If we need to do a negative rotation then the equations will be:

$$x_{i+1} = (x_i + y_i \tan \alpha_i)$$

$$y_{i+1} = (y_i - x_i \tan \alpha_i)$$

$$z_{i+1} = z_i + \alpha_i$$

Note that the constant value that was taken out doesn't change irrespective of positive or negative rotation.

Let's assume that $x_0 = X$, $y_0 = Y$ and the initial angle of the vector is Z then after "n" rotations we have:

$$x_n = X \cos\left(\sum_{i=0}^{n} \alpha_i\right) - Y \sin\left(\sum_{i=0}^{n} \alpha_i\right)$$

$$y_n = Y \cos\left(\sum_{i=0}^{n} \alpha_i\right) + Y \sin\left(\sum_{i=0}^{n} \alpha_i\right)$$

$$z_n = Z - \sum_{i=0}^{n} \alpha_i$$

When we consider the pseudo rotations (13), we have

$$x_n = K\left(X \cos\left(\sum_{i=0}^{n} \alpha_i\right) - Y \sin\left(\sum_{i=0}^{n} \alpha_i\right)\right)$$

$$y_n = K\left(Y \cos\left(\sum_{i=0}^{n} \alpha_i\right) + Y \sin\left(\sum_{i=0}^{n} \alpha_i\right)\right)$$

$$z_n = Z - \sum_{i=0}^{n} \alpha_i$$

Where:

$$K = \prod_{k=1}^{n} \sqrt{1 + (\tan \alpha_i)^2}$$

Now that we have the entire math in place we need to figure out the values for α. As we said before, to simplify the expression $\tan \alpha_i$ we need to choose α_i such that $\tan \alpha_i$

is a power of 2. The following table gives the various α value which stratifies the above condition.

Iteration number	**α_i (degree)**	**$\tan \alpha_i$**	**$\sqrt{1 + (\tan \alpha_i)^2}$**
1	45	1	1.414213562
2	26.56505118	0.5	1.118033989
3	14.03624347	0.25	1.030776406
4	7.125016349	0.125	1.007782219
5	3.576334375	0.0625	1.001951221
6	1.789910608	0.03125	1.000488162
7	0.89517371	0.015625	1.000122063
8	0.447614171	0.0078125	1.000030517
9	0.2238105	0.00390625	1.000007629
10	0.111905677	0.001953125	1.000001907
11	0.055952892	0.000976563	1.000000477
12	0.027976453	0.000488281	1.000000119
13	0.013988227	0.000244141	1.00000003
14	0.006994114	0.00012207	1.000000007
15	0.003497057	6.10352E-05	1.000000002
16	0.001748528	3.05176E-05	1

Table 1: Rotation Iteration number and corresponding values

Using the above α_i we can produce any rotation angle we need. During the design stage we need to fix the number of rotations that will be performed. The accuracy is improved as the number of iterations increases. The number of rotations has to be fixed so that *k* becomes a constant and correction factor can be applied to the result at the end of the calculation. For example let us choose the number of iteration as 8 and if we need to achieve a 30 degree rotation then what to follow is set of angles that will be added/subtracted:

$$30.26486 = 45 - 26.56505118 + 14.03624347 - 7.125016349 + 3.576334375 + 1.789910608 - 0.89517371 + 0.447614171$$

It can be seen from the equation above, that the targeted 30 degrees resulted in an angle of 30.26486. If the number of iterations are increased then the accuracy increases but the system becomes slower to calculate the value. It should be noted that the intended angular rotation might be achieved within the first few iterations but since we need to keep the *k* value constant we do all the designed iterations.

The value of the constant "*k*" for 16 iterations is 0.607253, for infinite number of iterations 0.6072529350088812561694 (13).

4.3 Digital perspective

As we mentioned before by using the values of α for the iterations we reduce the value of $\tan\alpha_i$ to a power of two. Incorporating this and the rotation direction we can rewrite the above equations as:

$$x_{i+1} = x_i - d_i\left(2^{-i}y_i\right)$$

$$y_{i+1} = y_i + d_i\left(2^{-i}x_i\right)$$

$$z_{i+1} = z_i - d_i \tan^{-1} 2^{-i}$$

where d_i is the direction of rotation. The value for $\tan^{-1} 2^{-i}$ is generally looked up from a ROM as it is a constant for a particular rotation.

4.4 Modes of operation

A CORDIC processor can work in two modes of operation, namely rotation mode and vectoring mode. In rotation mode of operation, the intent is to converge the value of z_i to zero. The vector lies on the x axis, that is y_i is zero, and the z_i is loaded with the angle by which the vector needs to be rotated. We go through the iterative process to make z_i zero. The direction of rotation is determined by the sign of the value of z_i. After the rotation value of x_i gives $\cos z_i$ and the value in y_i gives $\sin z_i$.

In case of vectoring mode, the vector is in the first quadrant and z_i is loaded with zero. The intent is to converge the value of y_i to zero. Hence the direction of rotation is determined using sign bit of y_i. If y_i is positive then the direction of rotation is negative and if y_i is negative then the direction of rotation is positive. As we go through the rotation process, z_i value accumulates the angular rotation being performed. At the end of the process x_i contains the magnitude and z_i contains the initial angle of the vector. For our application we use the CORDIC in vectoring mode.

CHAPTER 5: ARCHITECTURE

This chapter describes high level and low level architecture of the digital sub-system in detail. The digital sub-system is divided into eight functional units, namely Decimation filter, Calibration unit, CORDIC CORE, Scalar unit, I^2C unit, Clock generator and system controller. Following is the high level block diagram of the digital sub-system:

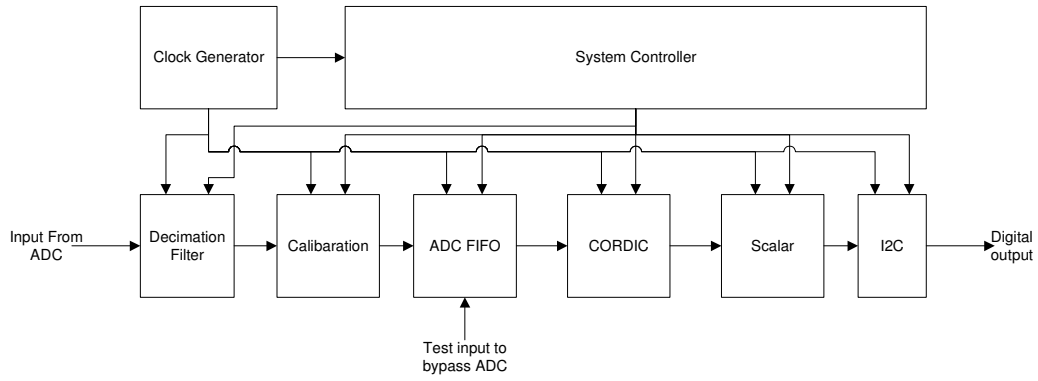

Figure 2: The Digital Sub-System

Each functional unit acts like a pipe stage, which allows us to run the system in a sequential mode or in pipelined mode. To enable this we define a system clock and each functional unit completes its operation within one period of this clock. The functional units in most cases require more than one clock cycle, hence each unit runs at different frequency depending on its requirement. To enable easy data transfer without meta-stability the clocks are designed in such a way that they have a synchronized positive edge with each other and the system clock. That is when there is a positive edge in the system clock; all the clocks in the system have a synchronized positive edge. This allows us to transfer data between functional units without synchronization issues. For additional protection the functional units are designed in such a way that they setup and hold data long enough for the receiving functional units to cleanly pick up the data and proceed.

The data format is chosen as 16 bit fixed point format. Hence when the data is transformed between two functional units, the data is 16 bits wide but the binary point is fixed based on the functional unit's requirement and the receiving unit is in sync with it. Within the functional unit the data might have to increase in the width depending on the operation and the algorithm. For example, the data width within the decimation filter is 32 and when the data is transferred out of the functional unit the higher 16 bits are transferred out. Similarly, in the scalar unit, the internal data size is 28 bits due to the multiplier. The upper 16 bits are transferred out and the rest discarded. There are no special overflow protections in the functional units. The overflow is taken care by design. Since we know the input range, the system is designed such that at no point the data will exceed the maximum range that can be handled by the number of bits used.

Area was one of the main considerations while designing the system. One of the effective methods to reduce area is to reduce the size of the datapath. Hence the minimum possible datapath width of 1 was chosen. To achieve this, all complicated structures like multipliers and dividers were avoided and serial adders are used wherever possible. All the complex operations, if required, were done using iterative serial addition.

To ease the data transfer between functional block, it is better to do it in one clock cycles. There are time domain crossovers between all the functional blocks. Serial transfer is not viable as the sending and receiving functional units are in different clock frequencies. Therefore the data transfers between the functional blocks are done in parallel. To achieve this, the shift registers that hold the data are all capable of giving parallel out data. We determine the clock periods during which the valid data is available and synchronize the receiving functional unit to pick up the data.

When the system runs in pipelined mode we can achieve a very high throughput. The system has 5 stages, the ADC, Decimation Filter, CORDIC unit, Scalar unit and I^2C

unit. When the system is in sequential mode of operation, one set of magnitude and phase data is generated for every 5 system cycle. The throughput is 1 out 5 cycles even though the front end Sensor and ADC are functional. To increase the throughput the system is pipelined. In the pipelined mode we get one set of data every system cycle. In this mode the frequency of operation is limited by the ADC speed.

5.1 Decimation Filter

5.1.1 INTRODUCTION

The sigma delta ADC in the front end gives a single wire pulse width modulated output. The output cannot be directly used as the data is embedded in the pulse width modulated signal. An example of the signal is as shown below (17).

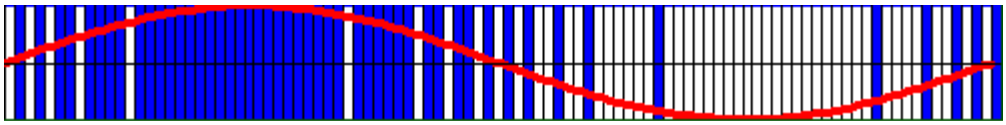

Figure 3: Sigma Delta ADC Output

The signal output from Sigma Delta is highly oversampled and contains heavy amount of noise in the higher frequencies. To extract information out of the signal we need some sort of low pass filter and a down-sampler to reduce the sample. A decimation filter is one of the filters that combine both these aspects. A decimation filter is a digital filtering technique for reducing number of samples in discrete-time domain and contains a low pass filter.

A decimation filter consists of two parts, a low pass anti-aliasing filter and a down-sampler. An anti-aliasing filter is a filter used to restrict the bandwidth of a signal suitably to satisfy the Nyquist-Shannon Sampling Theorem. One of the common implementation of the low pass anti-aliasing filter is a sinc filter. A sinc filter is a filter

that removes all frequency components above a given bandwidth. The impulse response of the filter is a Sinc function and the frequency domain response is a rectangle function.

Since the Sigma Delta output is an over sampled signal, we need a down-sampler. Down-sampling is a process of reducing the sampling rate of a signal; this helps in reducing the data rate of the input signal. Since the sampling rate is reduced, we need to a make sure that the Shannon-Nyquist sampling theorem condition is maintained. Otherwise we can have aliasing issues. To negate this possibility we use the anti-aliasing filter as described in the previous paragraph.

5.1.2 IMPLEMENTATION

Hence the over-sampled output of the sigma-delta ADC needs to be processed by a digital decimation filter shown below.

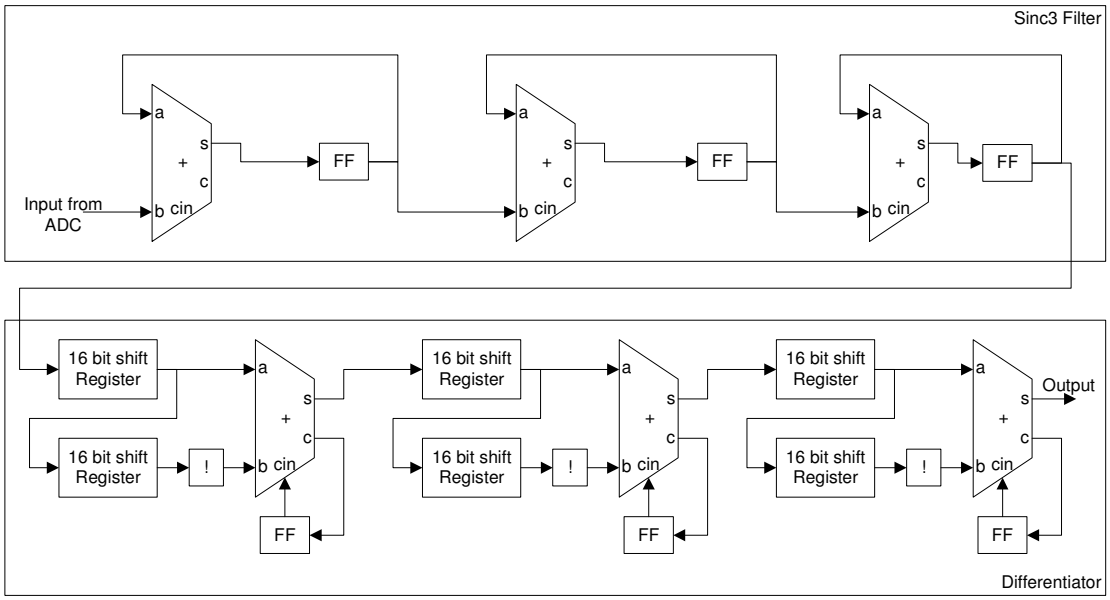

Figure 4: Decimation Filter, sinc3 Filter and Differentiator

The decimation filter consists of 6 stages. The first three stages perform the sinc3 filter function. The accumulators act like a sinc filter and filters out the high frequency

components of the input signal. The next 3 stages perform the differentiation function. The output of the final stage gives the digital data corresponding to the analog signal.

5.2 Calibration and buffer

5.2.1 INTRODUCTION

Since we are using CMOS process we don't have the notion of negative voltages. With the differential input we need to have the V_{cm} fixed at 0.9v instead of 0v. Hence when we have a differential input of zero the ADC actually works on input voltage of 0.9v. Hence we track this value and need to subtract it from the actual ADC data output from the ADC to get the correct value.

When the system comes out of reset, the system enters calibration mode. This mode lasts for one system cycle. During this period we expect the input to ADC fixed at 0.9v and the ADC performs conversions for "I" and "Q" channels once. When the data comes out of the decimation filter serially we load the data into the "I" and "Q" calibration registers respectively. We need to do the calibration for "I" and "Q" separately because there could be variation in the data coming out of the sensor for 0.9v. Once the calibration is done, before the data enters into the buffers the calibration value is subtracted serially out of the data.

Since we are sharing a single ADC for both the in-phase, I channel, and the Quadrature, Q channel, the data for I and Q comes serially. The CORDIC core though requires both the "I" and "Q" simultaneously, hence we need to buffer the "I" input coming from ADC till we have the "Q" data.

5.2.2 IMPLEMENTATION

Following is the block diagram of the calibration and the buffer unit.

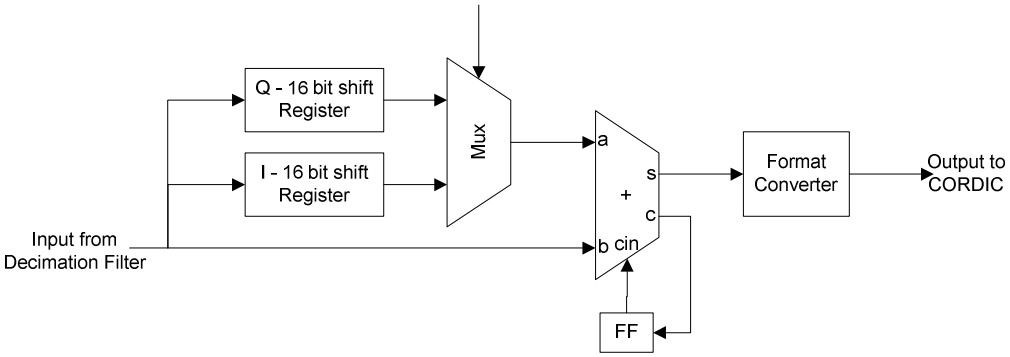

Figure 5: Calibration Block

When the data comes serially out of the decimation filter, it is loaded into the calibration registers during the calibration mode. The system controller indicates if that data corresponds to "I" channel or "Q" channel. Using this information we load the data into the correct calibration register.

During normal mode of operation, the calibration registers are put in a loop back mode and at the same time a multiplexer selects between the "I" calibration register and "Q" calibration register depending on the data being processed. The loop back makes sure the calibration data is not lost and the data coming out of the multiplexer is used to correct the data coming out of the decimation filter. The correction is a subtract operation and we use a serial adder for this purpose. A serial adder will serve the purpose as both the inputs and the serial bit stream.

5.3 CORDIC Core

5.3.1 INTRODUCTION

CORDIC is a relatively simple structure capable of calculating complex mathematical equations using iterative addition process. The hardware needs to

implement the equations from Chapter 4: CORDIC. The algorithm for the implementation is as shown below.

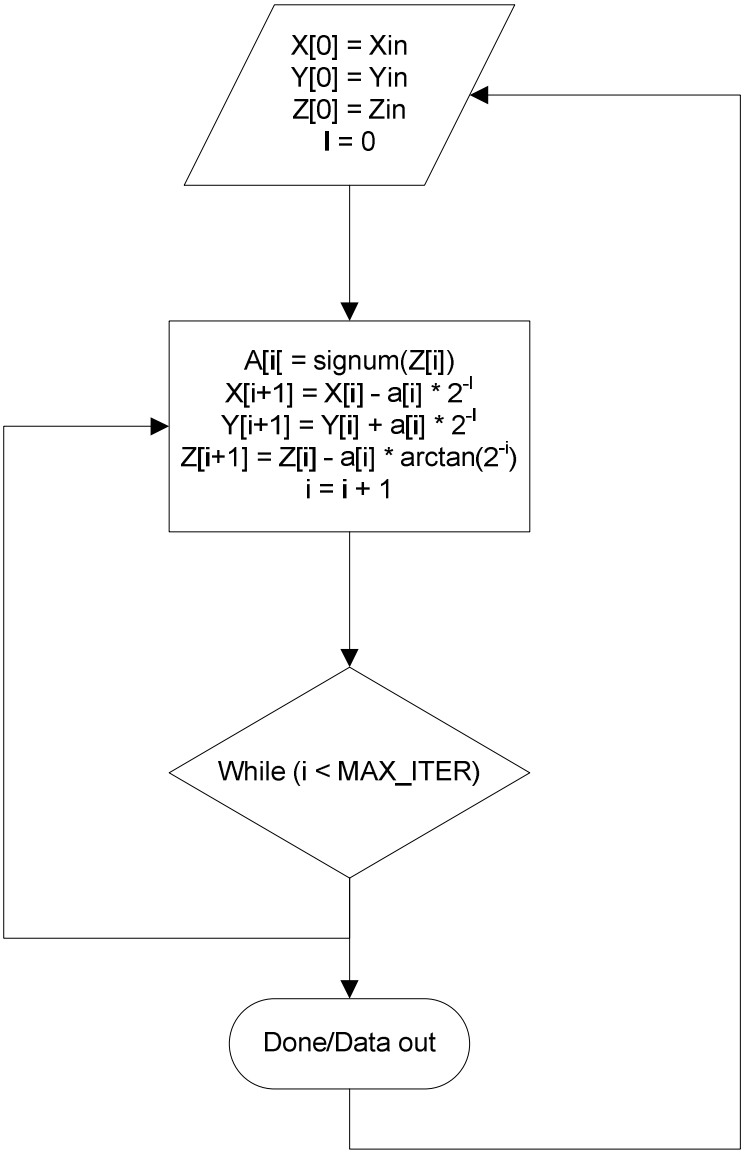

Figure 6: CORDIC Algorithm

5.3.2 TRADITIONAL CORDIC IMPLEMENTATION

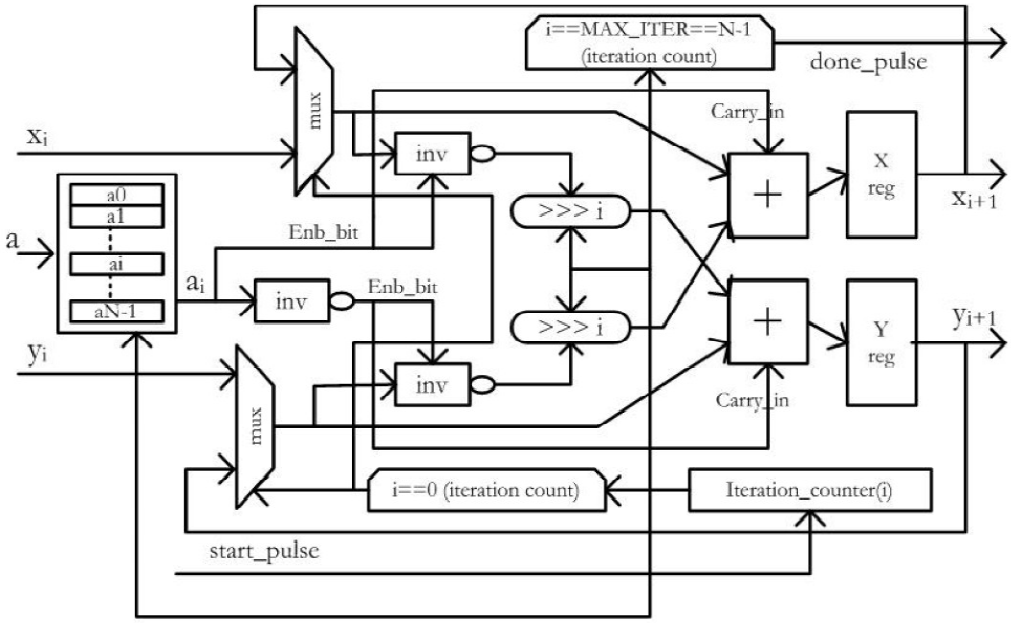

Figure 7: Traditional N recursive CORDIC structure

The figure above shows the traditional CORDIC implementation. In this implementation the datapath has three adders as wide as the number of bits in the system. Two adders compute the "x" and "y" values and the third adder computes the angle. The structure takes "n" cycles to compute the result where "n" is the number of iteration based on the accuracy that was decided.

5.3.3 OUR IMPLEMENTATION

Following figure shows the implementation that was chosen:

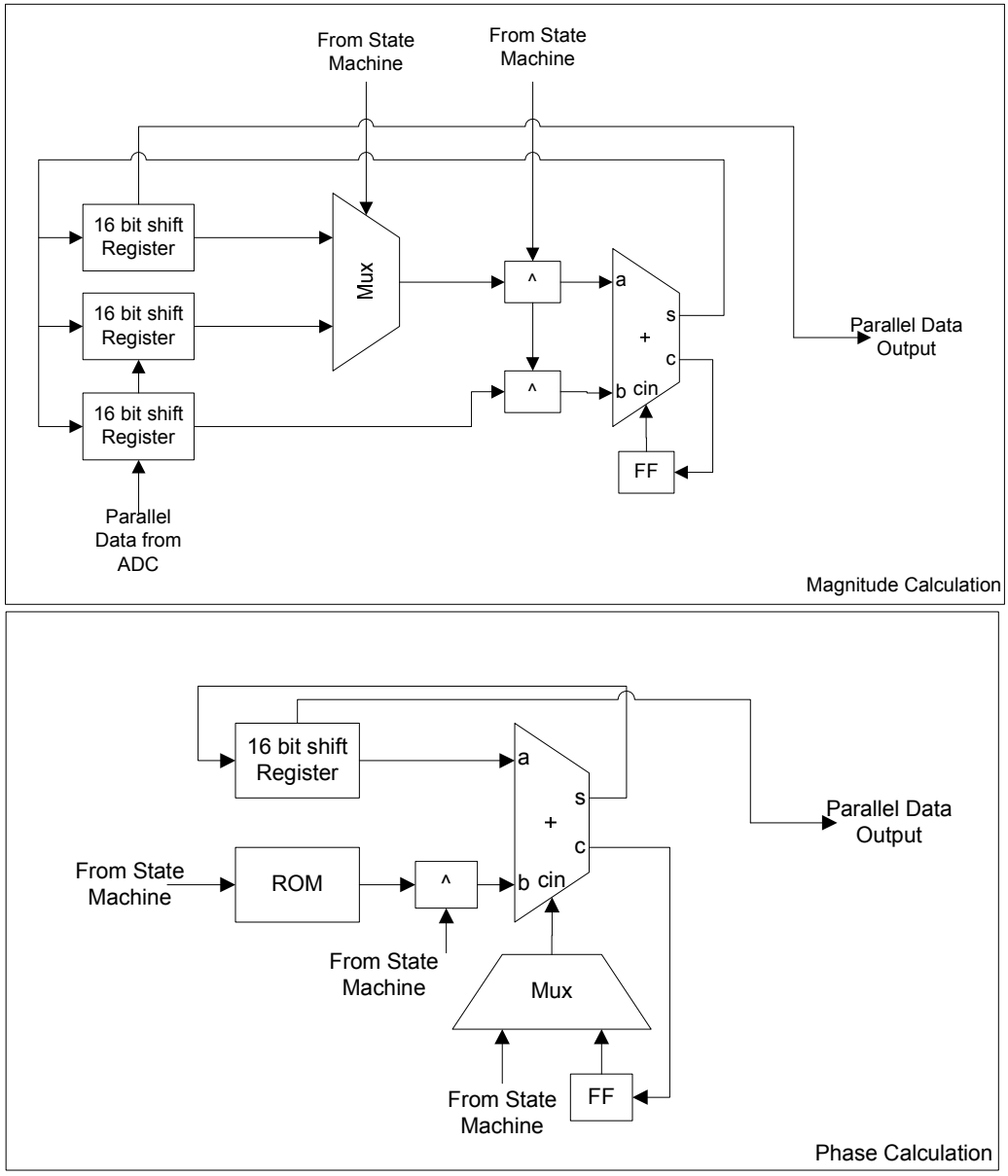

Figure 8: CORDIC Magnitude and Phase Calculation Logic

In this implementation, we have a single one bit adder. The adder is used as a serial adder for calculation of both "I" and "Q". The main idea is to first calculate the "I" channel and then calculate "Q" using the same adder. To preserve the value of "I" and to

enable the calculation of “Q”, the shift register for “I” is double buffered. This way we do not lose data of “I” channel, when we calculate the “Q” value.

In parallel we keep track of the rotation that is being performed. The fixed rotation angles are stored in a ROM. The algorithm dictates whether we need to add or subtract the angles from the accumulator.

5.3.4 THE CONTROLLER

Since we are using a single bit adder, we need quite a few control signals. Since it employs serial addition we need 16 cycles to perform the addition. In addition to this we need to perform 16 iterations. For each iteration, the calculations for “I” and “Q” are performed. Hence we will require at least 2 X 16 X 16 cycles to calculate the magnitude and phase values. Since the 16^{th} iteration for the “Q” is not used (the aim was to reduce this to zero), we really need not perform this calculation. Hence we need 496 cycles to perform the calculations.

Generating a clock that is 496 cycles faster than the system clock is difficult; hence we chose 512, since a clock that is a power of two is easily achievable. More over the extra 16 cycles can be used to transfer data in and out of the block.

The following is the state machine of the controller.

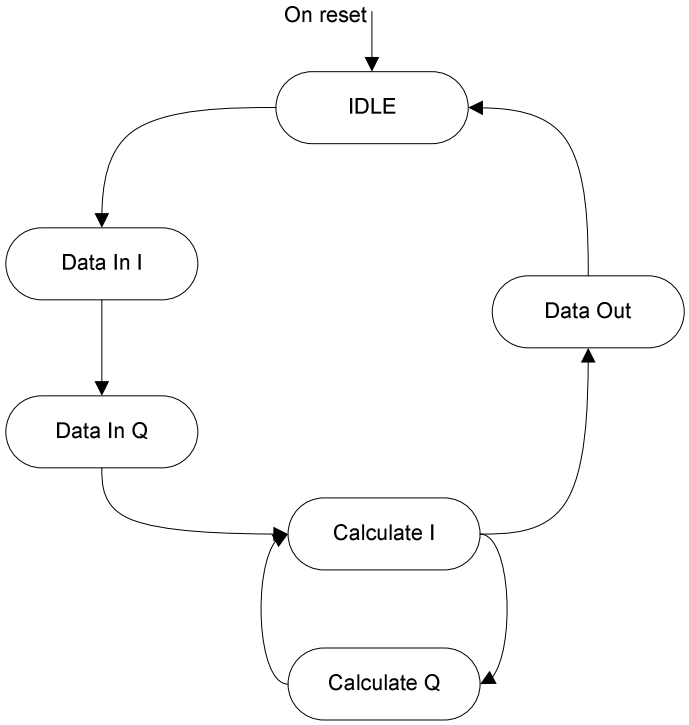

Figure 9: CORDIC Controller State Machine

State "Idle" is where the state machine is in restart state. The state machine is ready to receive the state signal from the system controller. Once the state machine gives the start signal, the state machine goes to the "Data In I" and then immediately to the "Data In Q" state. Where the data is loaded into the "I" and then "Q" registers from the FIFO respectively. Then the state machine moves to the "Calculate I" state where the next value of "I" is calculated. After 16 cycles the state machine moves to the "Calculate Q" state. Here the next value of "Q" is calculated. The state machine toggles between these two states 16 times; this is how the 16 iterations are performed. After the 16^{th} iteration of "I" the state machine moves to the "Data out" state. Here the data is held so that it can be consumed by the next stage.

The controller is implemented using a 9 bit counter. The counter starts counting once it receives the start signals from the system controller. The state traversal is achieved using the count values. That is when a predetermined count value is reached we assume that a state transition has occurred. Hence the count values themselves represent the states; we need not have a separate state registers. The control signals for the datapath are derived directly from the counter output.

5.4 Scalar unit

5.4.1 INTRODUCTION

As described in Chapter 4: CORDIC, we perform pseudo rotation during the computation to reduce the computation complexity. The error caused by the pseudo rotation needs to be corrected. The error is about 0.607253 and can be represented as 0x9B7 in hexadecimal. To do the correction we need to implement a multiplier that can multiply the above constant with the magnitude value coming out of the CORDIC unit. This correction is needed for the magnitude. The phase need not be adjusted as it doesn't carry any error. Since we need to keep the magnitude and phase in the same stages in the pipeline, the phase is buffered to compensate for the delay encountered by the magnitude value in this stage.

5.4.2 IMPLEMENTATION

The following figure shows the implementation of the scalar unit.

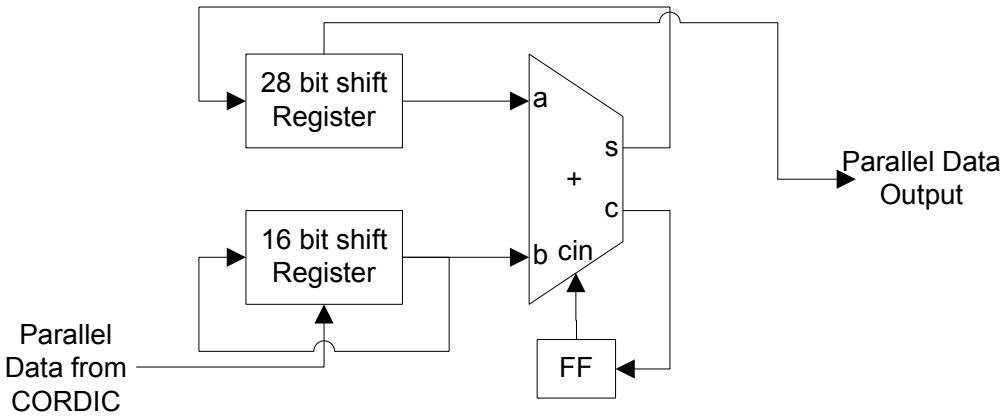

Figure 10: The Scalar Unit

The multiplier is implemented using a serial adder capable of adding 16 bit value. Form the constant we can see that there are about 7 "1" and each "1" represents an iteration that needs to be performed. 16 bits and 7 iterations would require 112 cycles. Since generating 112 cycles within a system clock period is difficult, we use a clock that has 128 cycles within one system clock period. The additional 16 cycles are used for transferring data in and out of the block.

There are two shift registers; the 16 bit register holds the output from the CORDIC unit and the 28 bit register is the accumulator. The 16 bit register is added to the specific regions of the accumulator and is stored back into the accumulator. The region to which the data is added is determined by the constant.

The constant has been determined to be 0x9B7 from Chapter 4: CORDIC. In total there are 8 "1", this would mean the process needs to be repeated 8 times consuming 128 cycles. This doesn't give any spare cycles for housing keeping activities like data transfer. Hence we approximate the value to 0x9B6 giving us 16 cycles for the house keeping activities.

Using this approximation, at the end of the seventh iteration the corrected value of the magnitude in the accumulator is passed on to the I^2C unit.

5.4.3 CONTROLLER

The following figure shows the state machine that controls the datapath.

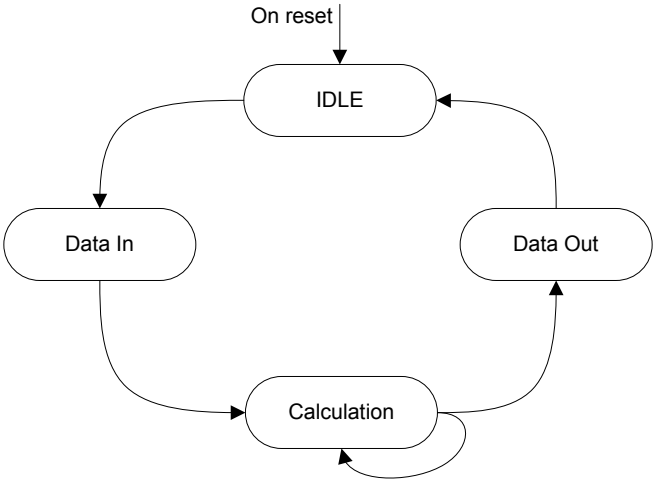

Figure 11: Scalar State Machine

The "Idle" state is the normal state in which the state machine is in after reset. Once the state machine receives the start signal the state machine moves the "Data in" state where the data is loaded into the 16 bit register from the CORDIC functional unit, at the same time the accumulator is cleared. Once the data is loaded the state machine moves to the "Calculation" state, where the multiply operation is performed. The state machine stays in this state for 112 cycles. Once the multiplication is completed the state machine moves to the "Data out" state. This data is ready to be transferred to the next functional unit, which is the I^2C unit. Finally the state machine moves back the "Idle" state, where it is ready to process the next set of data.

The state machine is implemented using a seven bit counter. We chose seven bits because it takes 128 cycles for the state machine to traverse completely and a seven bit

counter gives 128 states. We don't have separate registers to track the state of the state machine; instead we use the counter output directly. The value of the counter marks which state the state machine is in; this helps us to remove the additional registers that we would have required to maintain the states. The control signals for the data path are directly generated from the counter depending on the count value.

5.5 I^{2}C Functional Unit

5.5.1 INTRODUCTION

I^{2}C is a multi-master serial single-ended 2-wire computer bus invented by Philips. This protocol is widely used for inter-chip communications and is a relatively simple protocol to implement. The main advantage of the protocol is that it uses just two open drain wires for communication, SCL and SDA.

We need to transfer the calculated magnitude and phase data from the functional unit to outside the chip. Since we intend to operate the I^{2}C bus in a single-master single-slave configuration, we are simplifying the master to only send the data out without checking for contentions and other issues. Moreover, since we are not interested in reading data from any other device, we are fixing the master as a write only master and the address as 0x01.

5.5.2 TIMING DIAGRAM

Following is the timing diagram of I^{2}C (16)**.**

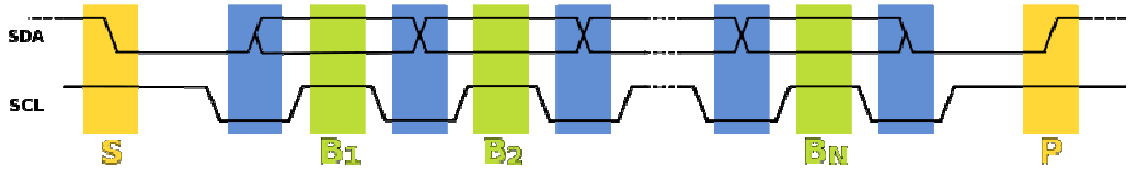

Figure 12: I^{2}C Timing Diagram

5.5.3 IMPLEMENTATION

The following figure shows the implementation of the I^2C master.

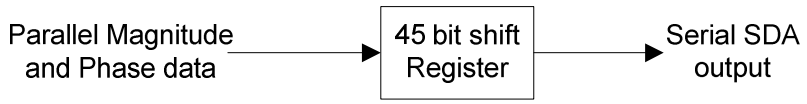

Figure 13: The I^2C Functional Unit Datapath

In total there are 32 data bits (16 for magnitude and 16 for phase) that needs to be transmitted. In addition to this we have 7 address bits and one read/write bit. The address is fixed to 0x01 and the read/write bit is set to 0x0 as the operation is always a write operation. We need to transmit the stop bit at the end of every byte. This brings the total number of bits to be transmitted to 45.

When the data is ready to be transmitted the magnitude and phase data is loaded into a 45 bit register at the appropriate bit positions. The register also contains all the other control bits like slave address, stop bits, etc. preloaded. When the data needs to be transmitted out, the register is put in shift register mode and data is shifted out in the required rate.

Since the number of bits to be sent out is 45, the nearest power of 2 is 64. This would mean we need to have 64 cycles within one system clock to transmit the data within one system clock. The timing requirement for the I^2C protocol is shown in the previous section. To satisfy the requirement, the functional unit takes in a clock that is 4 times the required clock frequency, which is 256 times the system clock. This clock is then appropriately divided to generate the clocks for the shift register, the SCL output and the controller.

5.5.4 THE CONTROLLER

Following figure shows the state machine of the controller.

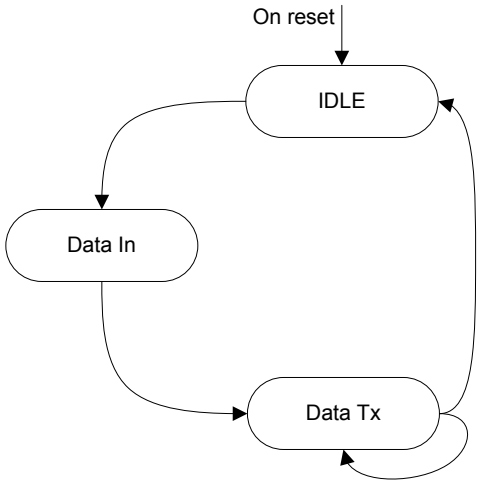

Figure 14: I^2C State Machine

The "Idle" state is the state in which the state machine enters upon reset. Once the state machine receives the start signal from the system controller the state machine moves to the "Data In" state. In this state the data from the Scalar functional unit is loaded. Once the data is loaded the state machine moves to the "Data TX" state. In this state the SCL output is enabled and the 45 bit shift register starts shifting data out on the SDA output. Once the data has been shifted out, the state machine moves back to the "Idle" state.

The state machine is implemented using a 6 bit counter and the count values represent the state of the state machine. The control signals for the datapath are directly generated from the counter.

5.6. Clock Generator

5.6.1 INTRODUCTION

The functional units require clocks at various frequencies. All these clocks needs to be in-phase with positive edge synchronized. To implement this we need a clock

divider circuit that can divide a single input clock and generate all the required frequencies. So far from the above discussion we know that we need the following frequencies:

- 1024 times the system clock, for Decimation filter
- 512 times the system clock, for CORDIC
- 128 times the system clock, for Scalar
- 256 times the system clock, for I^2C

In addition to these we need to generate non overlapping clocks for the ADCs. The timing diagram requirements for that is as shown below:

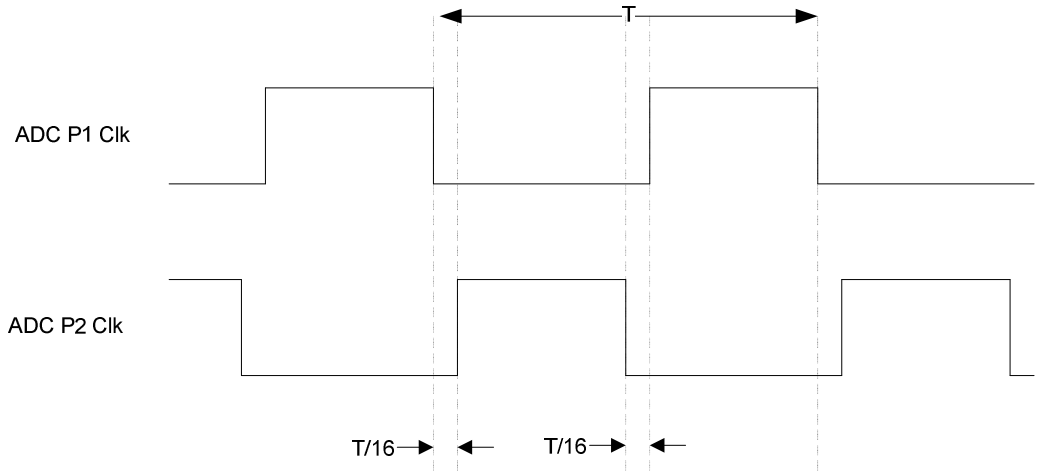

Figure 15: Timing Requirement for the ADC Clock

To generate these we need the input clock to be 2 to the power 17 times the system clock which is twice the ADC conversion period. The system clock is twice the ADC conversion period because we need to do two sets of ADC conversion for one set of data. We know that the ADC operates at 60 Hz. Thus the input clock frequency is 7,864,320 Hz.

5.6.2 IMPLEMENTATION

The divider is a 17 bit down counter. On reset the counter is loaded with 0x1FFF and when the system is out of reset the systems starts down-counting. The output of this counter is directly considered as the clocks for the ADC and various functional units. This ensures that the clocks are in phase and are positive edge synchronized. The clocks are the output of flip flops. This ensures that there are no glitches in the clock which otherwise can prove disastrous.

5.7 System controller

5.7.1 INTRODUCTION

As we have so far described there are quite a few functional blocks and we need a controller that can control the individual controller of the functional units. The main function of the system controller is to give the start signal for the individual controllers at the correct time.

As we described in the introduction there are two modes of operation, serial and pipelined. In the serial mode of operation each functional unit is activated one after the other in a serial fashion. During pipelined mode of operation the functional units are started one after the other till the pipeline is full. Once the pipeline is full, all the functional units operate in parallel thereby maximizing the throughput.

5.7.2 IMPLEMENTATION

Following is the state machine in case of the serial mode.

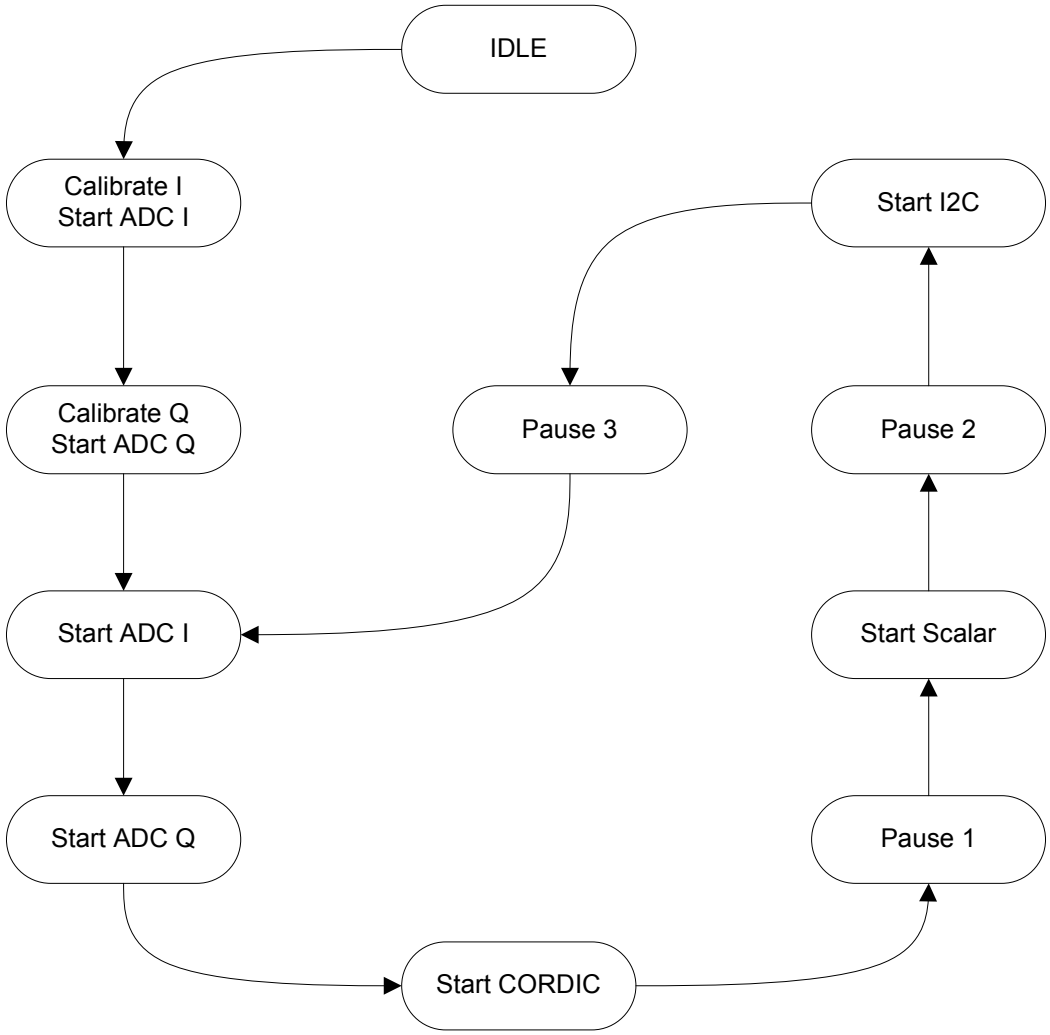

Figure 16: State Machine for Serial Mode of Operation

The "Idle" state is the state into which the state machine enters upon reset. The "Calibrate I; Start ADC I" state sets the system into the Calibration mode for "I". This sets the ADC to work on channel "I" and start the ADC. The next state "Calibrate Q; Start ADC Q" sets the system into the Calibration mode for "Q", sets the ADC to operate on "Q" channel and starts the ADC. Once these two starts have been processed, the system has completed calibration and the system is ready to process actual data.

The “Start ADC I” state, sets the ADC to operate on “I” channel and starts the ADC. The “Start ADC Q” state sets the ADC to operate on “Q” channel and starts the ADC. The state “Start CORDIC” starts the CORDIC functional unit. The start “Start Scalar” starts the Scalar functional unit. The state “Start I2C” starts the I^2C function unit. Since we have “I” and “Q” channels and a single ADC we need two cycles of ADC for every cycle of CORDIC, Scalar and I^2C functional unit. Due to this the controller clock is twice the speed of the system clock. This means that there needs to be a pause between starting the CORDIC and the Scalar functional units and a pause between Scalar and I^2C functional units and a pause between I^2C and the next ADC Start. These pauses are satisfied using “Pause 1”, “Pause 2” and “Pause 3” states.

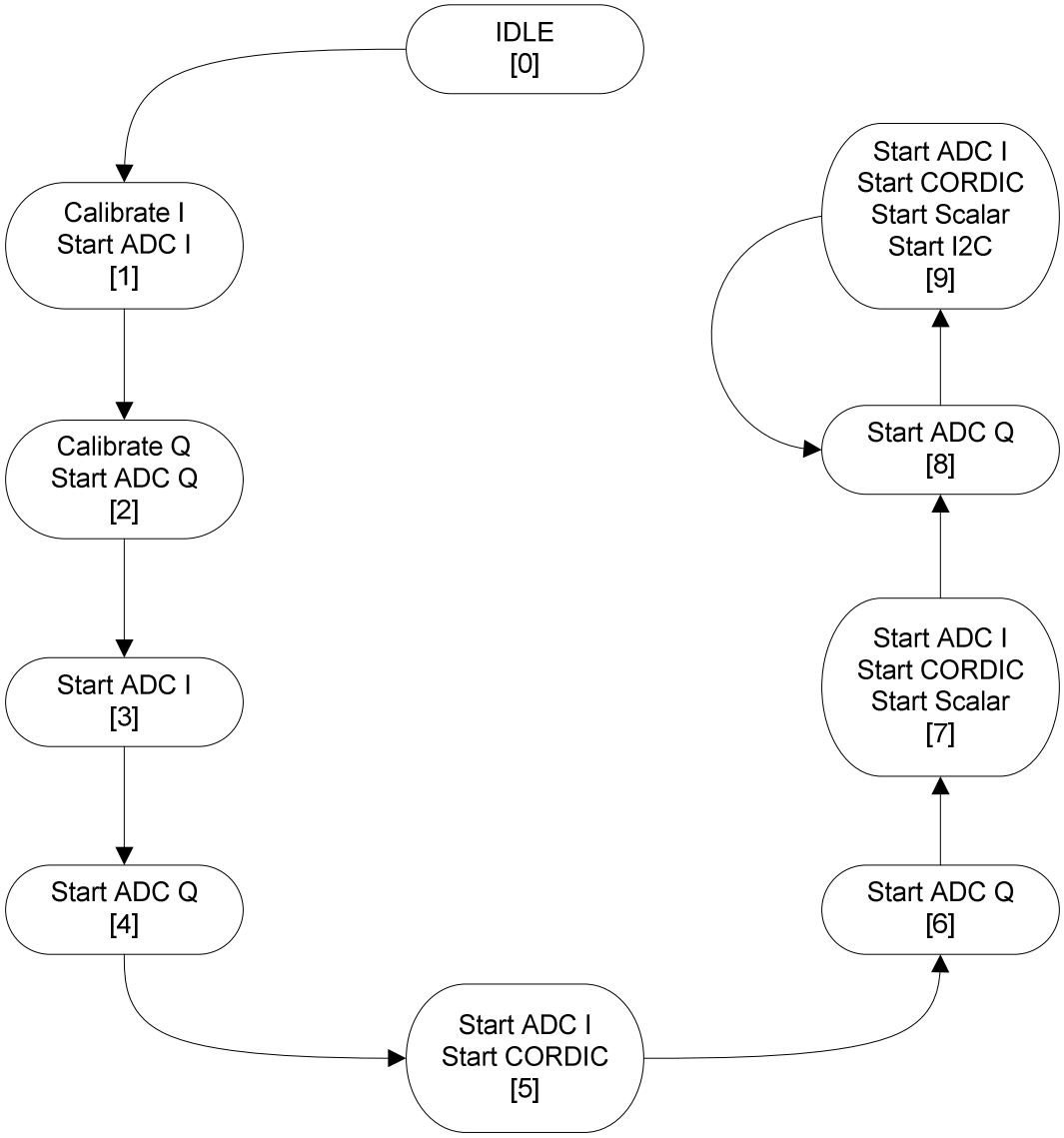

Figure 17: State Machine for Pipelined Mode of Operation

The above figure shows the state machine for the pipelined mode of operation. State 0 to Start 4 are same as the ones explained in the serial mode of operation. In State 5, we set the ADC on Channel "I" and start the CORDIC functional unit we have one set of "I" and "Q" data. In State 6, we set the ADC to operate on Channel "Q". Now we have data set 1 processed by the CORDIC functional unit and data set 2 out of ADC. Hence in State 7, we set the ADC to operate on Channel "I", start CORDIC and start Scalar

functional unit. In Start 8, we set the ADC to operate on Channel "Q". Now we have data set 1 through scalar functional unit, data set 2 through CORDIC functional unit and data set 3 out of ADC. With these data the pipeline is full and all the functional units can be activated at the same time. Hence in start 9 we start the ADC after setting it to operate on "I" channel, CORDIC functional unit, Scalar Functional Unit and the I^2C functional Unit. After this state the state machine moves to state 8 and remains in this loop.

The state machine is implemented using a ROM. Each entry in the ROM represents a state and the data coming out of the ROM gives the next address from which the control signals need to obtained and the current control signals. A structure of the ROM is as shown below:

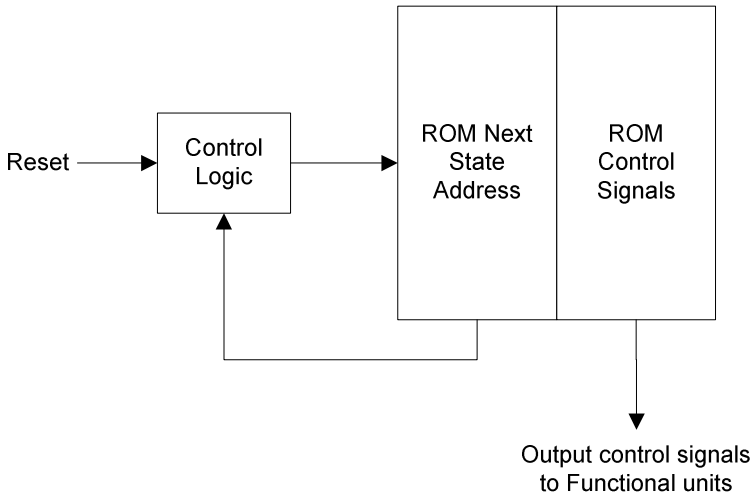

Figure 18: System Controller

5.8 Debug and Test Structures

5.8.1 INTRODUCTION

Debug and Test structures are necessary so that we can externally control the system and provide visibility into the system. Without the test structures it is difficult to test the chip as primary inputs and primary output do not give enough visibility.

The main aspects of debug are, control over the clocks and control over the data and the system controller output. If we can provide these aspects then we will be able to debug the digital system better.

To enable structural testing, the whole design has been scan inserted. All the units have been scan inserted except for the clock controller part, as we need the clock controller to provide test clock to perform the scan tests.

5.8.2 IMPLEMENTATION

As we have seen, the system has quite a few clocks. We can't control the ADC clocks as they are free running and analog system needs to provide this functionality. Since we are running ADC in free running mode, the decimation filter is also considered part of that system.

For every generated clock we provide two other clock options, the input system clock with the divider turned off and the system test clock TCK. A multiplexer is used to select between these clock sources. The control signals for the multiplexers are fixed and controlled by another test structure described later. Since the control signals don't change during the operation of the system, we will not need to bother about glitches.

The clock selection structure is as shown below.

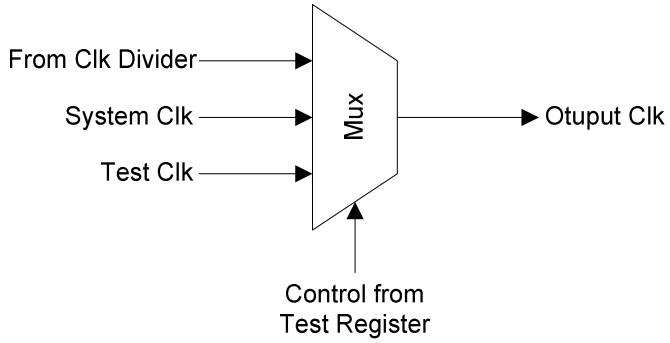

Figure 19: Clock Multiplexer used to Select Clocks

The best way to remove the dependence on the ADC to provide data would be to incept the ADC FIFO buffers and load them with the test data directly. To achieve this goal we have provided a one bit input through which we can shift in data serially into the FIFO flops. In this mode the TCK is provided as the clock to FIFO flops.

The system controller is responsible for scheduling data between various functional units and is responsible for generating the start signal for the functional units. It is quite important to control the output of this controller. To achieve this we have a multiplexer at the output of the Control Store ROM that selects between the control store output and an output from a test register that can be programmed using a TAP Controller.

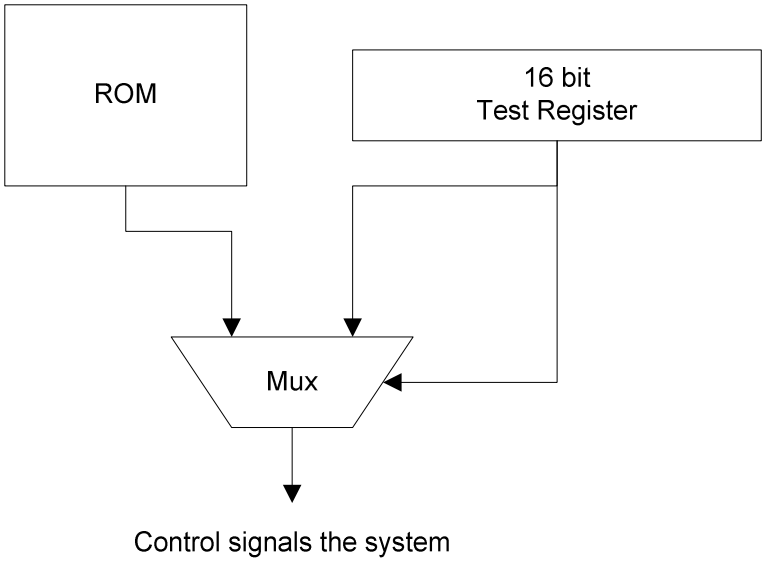

Figure 20: Test Structure to Control System Controller Output

The control signal for the multiplexer also comes from the test register. Since the test register is loaded using the TAP controller, it is essentially serial load operation. The problem with the serial load is that, when the data is being loaded into the register the parallel output from the register will change. This can mess up the operation of the system. To avoid this issue the test register is provided with a shadow register where the control signals are actually shifted in. Once all the control signals are shifted in we do a parallel load into the actual test register which supplies data to the multiplex.

The structure is as shown below:

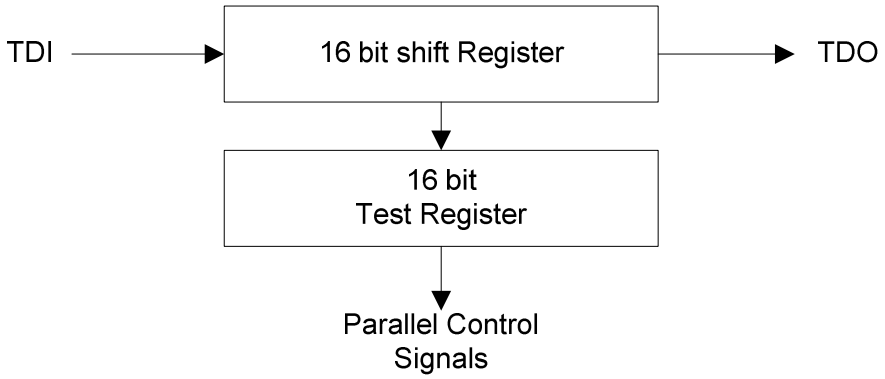

Figure 21: Test Register Structure

5.9 JTAG TAP Controllers

The JTAG TAP controller is an IEEE 1149.1 standard for the Test Access Port (TAP) and Boundary scan architecture. It is very widely used as Debug ports. In this IC we are using the TAP controller for performing the Scan based structural tests and for functional debug.

The system has one scan chain and one debug register. One address is allocated for each of these. Hence the debug register can be accessed by loading the appropriate code into the Instruction Register of the TAP controller. Similarly the scan chain can be accessed by loading the scan chain's address in the instruction register.

To enable the scan test, first the debug register needs to be configured in such a way that the system is running on the test clock. This is very important because, if all the flops in the system are not on the test clock then the scan chains will not work correctly. Once this is done we can access the scan chain by loading the instruction register of the TAP the scan chain code.

CHAPTER 6: IMPLEMENTATION

6.1 Register Transfer Language

In the previous chapter the design of each and every functional unit in the system was described. To translate this into hardware, we use Verilog hardware descriptive language to describe the functionality of each block. The hardware functionality can be described using Verilog at various levels, behavioral, register transfer logic, structural etc. We choose to describe the hardware design in Register Transfer Logic.

Verilog files were written for each of the functional units and are integrated using structural Verilog.

6.2 Simulation

Function Simulation was done using the Cadence Incisive tool package. Test bench was written in Verilog for each of the functional units and each of the units was tested individually for the functional correctness. Then a digital system level test bench was written with the capability to pump in sample Sigma Delta ADC output and checking the output of the digital system automatically. Since the test bench was able to check the results automatically, we were able to run regressions.

6.3 Synthesis

Once the system was verified for functional correctness, we used Design Compiler (DC) from Synopsys for synthesis. Since the design has multiple clock domains, we made sure that each of the clocks were correctly set.

We adopted a hierarchical two synthesis strategy. With the hierarchical strategy we first synthesis the individual functional units with two pass strategy. In the first pass the Register Transfer Level is read in and we compile with high effort. In the second pass

we enable scan insertion. With this pass we have a completely synthesized scan inserted netlist.

The individual netlists are read in along with the structural Verilog of the top level and we do a synthesis. After the synthesis the timing reports are generated and are analyzed for timing violations. The netlist generated is used for place and route.

6.4 Place and Route

We use Synopsys IC Compiler (ICC) as the automatic place and route tool. Since we really didn't have area or aspect ratio constraint, we set a utilization of 60% and set the aspect ratio as 1. Using these constraints the tool generated a floor plan.

Once the basic floor plan was built, we created the power rings and power rails. We assumed the power rails of width 1 um and spacing between the rails as 10 um. With these assumptions the rings and rails were built. The rails and the rings were built on metal 3 and 4. Metal 1 had power rails that connected the standard cell's power rails using the correct-by-construction methodology.

The design have quite a few clocks, and all the clocks needs to be in-phase as there are interactions between almost all clock domains. Even though the design takes care of data crossing time domain, is it a good idea to make sure the edges match up. For this purpose we define all the clocks and perform Clock Tree Synthesis (CTS) on the clocks. This produces a clock tree for all the clocks and the clocks are buffered depending on the fanout.

We decided to give metal 5 and metal 6 for integration with the analog components. Hence the router was restricted to use metal 2, 3 and 4. We do a two pass routing; during the first pass we make the router route along with high fanout synthesis. In the second pass we do an incremental route so that it can fix some scenic routes. Once

the route is done, we enable hold fixing and do another round of routing. This step inserts hold fixing buffer and solves hold issues.

In order to meet the metal density rules we perform metal fill using the Hercules DRC engine. Metal fill is performed on metal layers 1, 2, 3 and 4. In addition to this we insert fill cells; this helps to increase metal one density and helps with the ploy density.

Finally before we generate the GDS2 we perform LVS and DRC. These two steps are very important as we need to make sure the place and route tool didn't change the functionality and all the manufacturing design rules are met. In addition to this we write out a Verilog netlist and perform a formal verification with the RTL that was used to describe the design. This makes sure that the tools did not mess with the functionality of the design. Following is the image of the place and routed database.

6.5 Cadence

We decided to do the final GDS2 generation out of The Cadence Virtuoso environment. The analog front end design and layout is done in the Virtuoso environment. As described about the digital design "place and route" is done in ICC and then the GDS2 is generated.

To integrate these two designs, we read in the netlist generated out of ICC into the Virtuoso environment. We then stream in the GDS2 generated into the Virtuoso environment. We run a round of LVS between the read in netlist and the streamed in GDS2. This will catch any issues with the stream in operation. The digital design is then instantiated in the analog components and we place the streamed in place and routed digital design and complete the connections with the analog design and the IO ring.

We then perform a LVS of the complete design followed by DRC for the complete design. This ensures that the layout confirms to the schematic/netlist and the design is DRC error free.

Following is the image of the full chip layout.

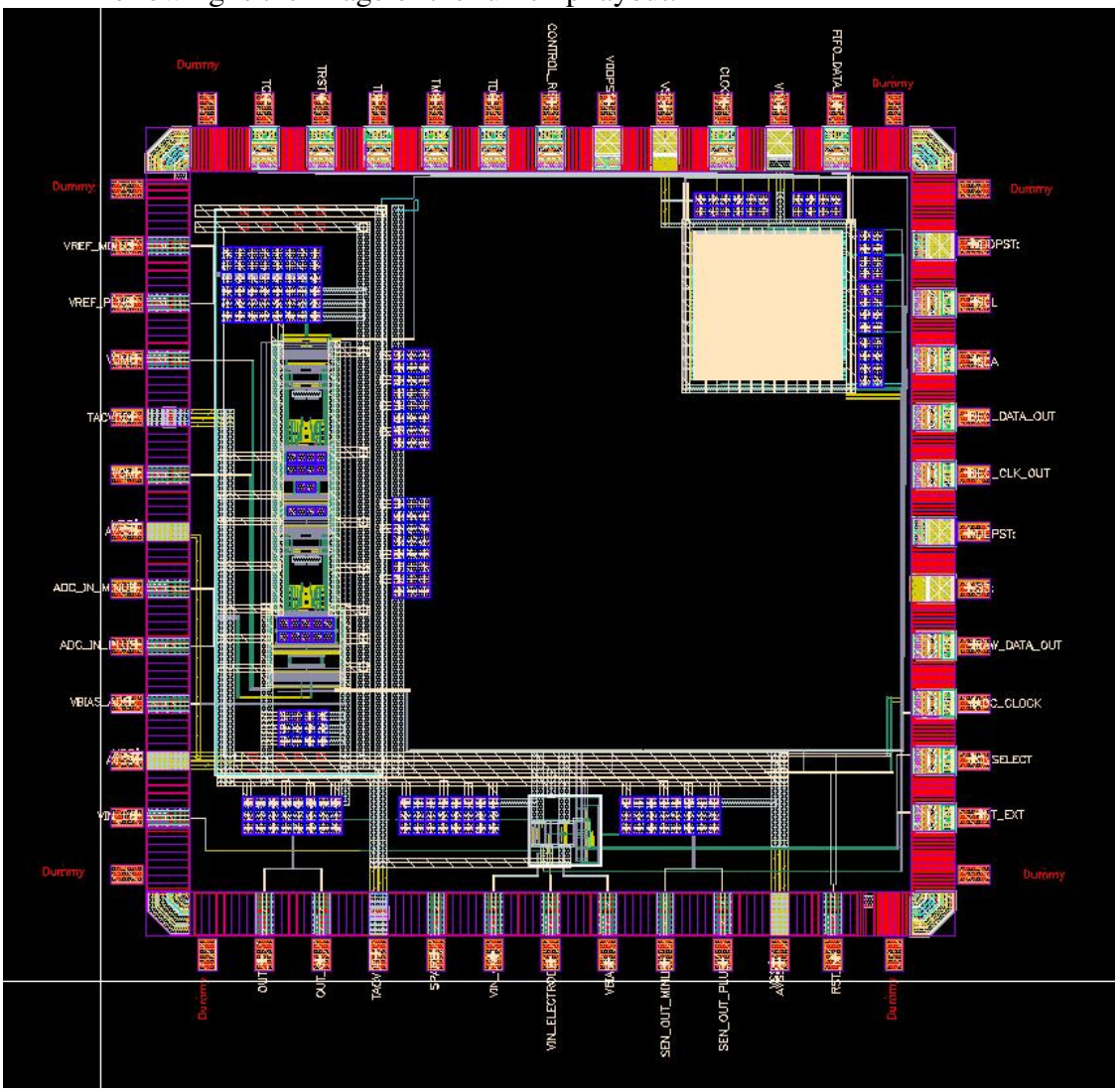

Figure 22: Full Chip Layout

Following is the Die photo of the Impedance Measurement Device from TSMC

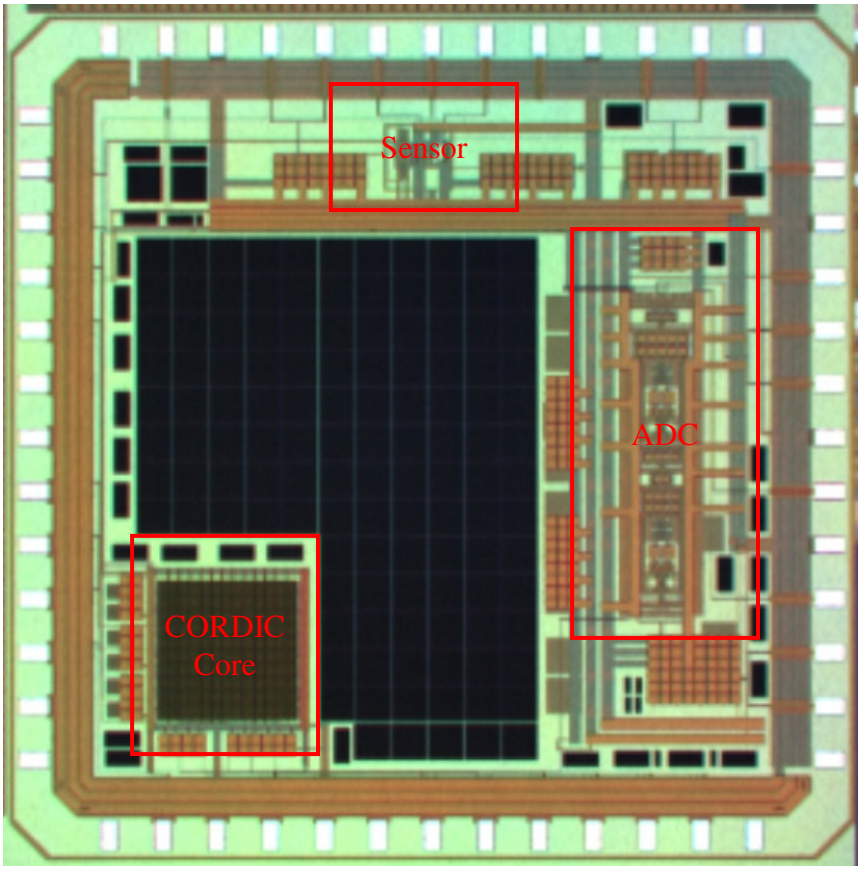

Figure 23: Die Photo from TSMC

CHAPTER 7: SILICON TESTING

To test the chip we need to build some kind of interface mechanism so that we can give in the stimuli and capture the resultant data. For this purpose we build a Printed Circuit Board (PCB). The PCB designed is a two layer board.

The test setup is as shown below.

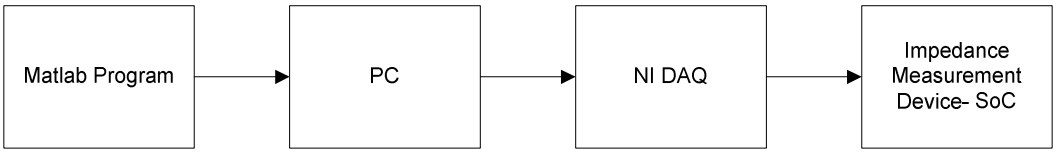

Figure 24: Test Setup

The Matlab program instructs the NI DAQ card to apply the appropriate stimuli. Once the stimuli are applied, the SoC starts generating outputs that are then connected to the NI DAQ card. The NI DAQ card samples the data which is made available to the Matlab program that post processes the data into a user friendly format.

The results obtained so far shows that the digital output linearly tracks the input for the differential input range of -0.7V to 0.7V. We suspect that the Scalar unit within the system is not functioning correctly. As a result, the output in this region is scaled by a factor of 0.57.

For the input range of -1.8V to -0.7V and 0.7V to 1.8V the system shows some non-linearity. We again suspect the Scalar unit for this, as it is giving a scale for the range of -0.7V to 0.7V; further debugging remains to be done.

It can be seen that the output from the digital system is highly repeatable for input frequencies ranging from 50 KHz to 2 MHz.

We are currently not able measure the phase as we have not yet switched on the sensor unit.

CHAPTER 8: CONCLUSION

This thesis has allowed for insight into the various aspects of chip design, and has touched on the areas of specification, design, implementation, simulation, P&R, integration, and silicon testing.

This thesis was a challenging yet rewarding experience. Limited testing resources meant conservative design with limited risks. We chose to use Flip Flops over RAM or ROM as we didn't have confidence in the ROM/RAM generators. A limited number of IO pins also meant a non-programmable control store.

The tools themselves also provided a significant challenge. The design had both analog and digital components. The analog component was designed in Cadence Virtuoso environment and the digital component was designed in Synopsys DC/ICC environment. Using tools from different vendors like Synopsys and Cadence meant having to specify design rules that satisfy both the tools. Another problem we had was transferring the digital design from Synopsys DC/ICC into the Cadence Virtuoso environment since the integration needed to be done in the Cadence Virtuoso environment.

Overall the thesis was a good learning experience specifically with respect to various tradeoffs involved in chip design.

Bibliography

1. *Principles of affinity-based biosensors.* **Rogers, Kim R.** 2000, Molecular Biotechnology, pp. 109-129.

2. *A CMOS Fluorescence-based Biosensor Microarray.* **Jang, B., et al.** s.l. : Tech. Dig. of International Solid-State Circuits Conference, 2009.

3. *A Fully Electronic DNA Sensor with 128 Positions and In-pixel A/D Conversion.* **al., M. Schienle et.** 2, Dec 2004, IEEE J. Solid-State Circuits, Vol. 39, pp. 2438-2445.

4. *Active CMOS Sensor array for Electrochemical Biomolecular Detection.* **Levine, P. M., et al.** 8, Aug 2008, IEEE J. Solid-State Circuits, Vol. 43, pp. 1859-1871.

5. *Fully Electronic CMOS DNA Detection Array Based on Capacitance Measurement with On-Chip Analog-to-Digital Conversion.* **Esposti, C. S. D., Guiducci, C. and al., C. Paulus et.** 2006. ISSCC Dig. Tech. Papers. pp. 48-49.

6. *A Frequency-Shift CMOS Magnetic Biosensor Array with Single-bead Sensitivity and No External Magnet.* **Wang, H., et al.** 2009. ISSCC Dig. Tech. Papers. pp. 438-439.

7. *Present and future of surface plasmon resonance biosensors.* **Homola, J.** 3, Oct 2003, J. Analytical and Bioanalytical Chemistry, Vol. 377, pp. 528-539.

8. *A CMOS Electrochemical Impedance Spectroscopy Biosensor Array for Label-free Biomolecular Detection.* **Manickam, A., et al.** 2010. Tech. Dig. of International Solid-State Circuits Conference (ISSCC).

9. *Unpowered resonant wireless sensor nets for structural health.* **Pasupathy, P., Zhuzhou, M. and Neikirk, D. P.** SL Wood : s.n., 2008. IEEE Sensors.

10. **Manickam, A.** *Integrated Impedance Spectroscopy for Biosensor Arrays.* Electrical and Computer Engineering, University of Texas at Austin. Austin : s.n., 2008. Thesis.

11. *The design of sigma-delta modulation analog-to-digital converters.* **Boser, BE and Wooley, BA.** 1988, IEEE Journal of Solid-State Circuits.

12. *The CORDIC Trigonometric Computing Technique.* **Volder, Jack E.** 1959. IRE Transactions on Electronic Computers. pp. 330-334.

13. CORDIC. *Wikipedia.* [Online] http://en.wikipedia.org/wiki/CORDIC.

14. *A Unified Algorithm for Elementary Functions.* **Walther, John S.** 1971. Proc. of Spring Joint Computer Conference. pp. 379-385.

15. *Pseudo Division and Pseudo Multiplication Processes.* **Meggitt, J. E.** April 1962, IBM Journal.

16. **Muller, J.-M.** *Elementary Functions: Algorithms and Implementation.* 2. Boston : Birkhäuser, 2006. p. 134.

17. I²C. *Wikipedia.* [Online] http://en.wikipedia.org/wiki/I%C2%B2C.

VITA

Muralidharan Jagannathan is a Masters student in Electrical and Computer Engineering Department of University of Texas at Austin. His area study includes VLSI, Analog Integrated Circuit Design and Micro-architecture. He completed his undergraduate program in Electronics and Communication in 2004 from Visvesvaraya Technological University.

Permanent address (or email): 4409, Koncept, Nandi Enclave
Bhuvaneswari Nagar, Banasankari 3rd Stage
Banagalore, 560 085
Karnataka, India

This thesis was typed by Muralidharan Jagannathan.