A Digital Demodulator for Frequency Modulated Signals

Yu Fubing

School of Computer Engineering

A thesis submitted to the Nanyang Technological University

in fulfilment of the requirement for the degree of

Master of Engineering

2005

Acknowledgements

First and foremost, I would like to express my profound gratitude to my supervisor Dr. Edmund Lai for his guidance, patience and encouragement. It has really been a privilege working under his supervision.

I wish to thank Dr. T. Srikanthan for providing me the opportunity to work in the Centre for High Performance Embedded Systems (CHiPES). I would also like to express my gratitude to the members of CHiPES for their kindness and co-operation. In particular, I take this opportunity to convey my sincere gratitude to Dr. Chang Chip Hong for numerous constructive suggestions offered to me. I have gained substantially from his assistance and his willingness to share his expertise. I also thank Ms. Nah Kiat Joo and Mr. Rande Heng for their logistic support and willingness to help.

I want to thank Ms. Tian Hui, my wife, who studied at CHiPES also and helped me to settle everything down in the School and gave me a lot of support and guidance on how to do a research project.

I would like to thank Infineon Technologies Asia Pacific Pte. Ltd. for allowing me to do this part-time research study. I also wish to acknowledge the support received from Mr. Liu Chong Hsu and Mr. M. Suraj who inspired me to pursue higher studies at NTU.

Finally, I would like to acknowledge the School of Computing Engineering, Nanyang Technological University for providing a very good and harmonious research environment and advanced research facilities.

Dedicate

To

my dearest wife, Tian Hui and my princess daughter, Yu Yutian

who brighten my life wonderfully

Table of Contents

List of Figures

List of Tables

Abstract

The trend of modern communication system design is to implement as much functionality as possible by digital methods. This means that most of the functionalities that are previously handled by analog electronics are now handled by digital solutions, for example, DSP algorithms via software. Some analog systems migrating to digital ones need fully hardware solutions. The work presented in this thesis is concerned with the design of a fully-digital FM demodulator on FPGA/ASIC.

Frequency modulation is widely used in analog, operational, terrestrial, microwave and satellite systems. A typical digital FM demodulator consists of ADCs, quadrature mixer, FM demodulation, low-pass filters and DACs. In which, quadrature mixer, FM demodulation and filters are implemented using software in DSP processor currently. To replace these functions with hardware implementation, many factors need to be taken into consideration. Considering the hardware architecture, mixed demodulation algorithm is chosen with better signal quality and robust performance according to simulation results.

The key components in a full digital mixed demodulator include the quadrature mixer, the mixed demodulator and the filters. The algorithms for quadrature mixing and mixed demodulation involve many trigonometric calculations such as sine, cosine, arctangent. To map these operations into hardware, they are conventionally performed at design-time and results are stored in lookup tables for later use. The resolution is therefore limited by the size of the table. An alternative approach is to make use of the CORDIC algorithm which provides the necessary trigonometric values at runtime. Both conventional CORDIC and modified CORDIC for the digital demodulator are surveyed. Modified CORDIC processor is implemented in mxied demodulator. Implementing the modified CORDIC algorithm for arctangent values saves divide operation for I to Q ratio in mixed demodulator.

The low pass IIR filter is used for outputs of quadrature mixer and demodulator

in the digital FM demodulator. To implement a digital filter, many approaches are proposed previously after a literature search. Most of them are targeted for DSP processing solution since DSP processor is wide-used in communication system.These DSP processors are capable of carrying out MAC operations, but have bandwidth limitation. Because DSP processors are sequential in nature, one operation can be performed on a single set of data at a time. Take the advantage of FPGA/ASIC architecture, co-efficient component is introduced. This structure is not only very flexible to adjust the coefficient, but also is easy to construct a parallel-pipelined architecture. Further, transposed form filters based on co-efficient components are built to work in the digital FM demodulator system. These FPGA based filters implemented with parallel-pipelined architecture further enhance the overall performance of the digital FM demodulator.

To achieve the results of CORDIC and low pass IIR filters in the limited runtime, frequency multiplication is required. In this case, PLL and DLL are two options. The DLL is used since it is a pure digital design with delay line structure. It is easier to integrated in a full digital hardware architecture.

A FPGA implementation is adopting above substructures. Simulation and experiment results show the performance of algorithm chosen and hardware implementation. Comparing with the same implementation in DSP processor, the FPGA implementation can achieve the same signal quality. Although robustness is slightly worse in FPGA, the processing speed is two times faster than DSP processor implementation. This is due to the pipelined structure in hardware.

Digital FM demodulator in FPGA provides another approach to replace the current analog FM receiver system, especially for the system requiring fast data processing. The same solution is also applicable to modulator.

Chapter 1

Introduction

1.1 Background and Motivation

Many analog communication systems are still being used today. One of the most popular analog modulation techniques for wireless communication is frequency modulation (FM). It is used, for example, for very high frequency (VHF) radio broadcast and for private mobile radio (PMR) which is commonly used by organizations like the police, fire departments, railroad or power supply companies for mobile communication. Frequency modulation is widely used in most analog, operational, terrestrial, microwave and satellite systems [13]. The wide usage and unprecedented growth in mobile communications have placed stringent demands on the performance, size and cost of FM demodulator in communication systems [43]. In order to improve the reliability of these systems, the trend in the implementation of modern communication systems is to replace as much of the analog circuitry as possible by digital ones.

One popular way to implement fully digital demodulators is by software running on digital signal processors (DSPs). This approach has the advantage of being flexible – altering functionality implies only a change in the software. However, in order to achieve the best performance to cost ratio, a flexible hardware solution is needed. Field Programmable Gate Arrays (FPGA) are able to provide high-performance

and flexible solutions to demanding signal processing applications.

In this thesis, we explore the design alternative issues involved in the implementation of fully digital FM demodulator. The overall architecture of the demodulator is shown in Figure 1.1. It mainly consists of low pass filters, a quadrature mixer and the mixed demodulator itself. The development of these hardware modules for realizing the mixed demodulation algorithm is of the interest in this research work. A fully-digitized hardware solution is developed and its performance is evaluated in comparison with a software solution using DSP.

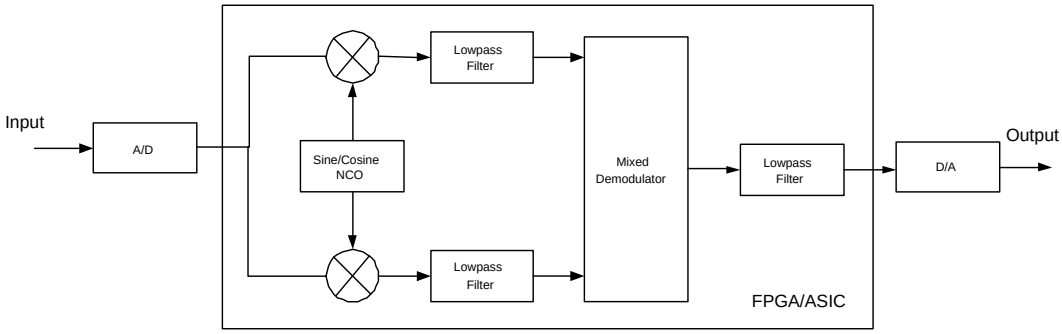

Figure 1.1: System architecture of FM mixed demodulator

1.2 Major Contributions of the Thesis

The major contributions of this thesis are listed as follows:

1. The mixed demodulator algorithm is implemented on a Xilinx FPGA. This algorithm provides a better Signal to Noise Ratio (SNR) verified by the FPGA implementation. The same signal quality as a software DSP implementation is achieved. Moreover, the throughput is twice that of the DSP processor implementation. (Chapters 4 and 5)

2. The coordinate rotation digital computer (CORDIC) algorithm is proposed to compute the arctangent values. A fully functional CORDIC algorithm processor is implemented including rotation mode and vector mode. The bit

parallel iterative CORDIC processor is chosen for its speed. The CORDIC algorithm used further optimizes the hardware architecture. A divide operation for I to Q ratio in mixed demodulator is saved. On the other hand, due to its iterative nature, frequency multiplication is required in a single master clock source system. (Section 4.3.1)

3. Frequency multiplication is achieved by using delay locked loop (DLL), a pure digital design with delay line structure. The fully digital design will not be affected by the manufacturing technology migration.(Section 4.2.2)

4. A transposed low pass filter is implemented in the hardware design. To take the advantage of FPGA architecture and keep the flexibility, co-efficient component is introduced and applied. FPGA based low pass filters are implemented with parallel-pipelined architecture, enhancing the overall performance.(Section 4.2.1)

1.3 Organization of the Thesis

This thesis is organized into six chapters with each chapter dedicated to a specific topic. The remaining chapters are organized as follows.

Chapter 2 provides an introduction to FM demodulation and a survey of major approaches for implementing a digital FM demodulator, including infinite impulse response (IIR) low pass filter, quadrature mixer and demodulator itself. CORDIC algorithm and its implementation are explored also.

Chapter 3 presents the mixed demodulation algorithm that has been adopted in the work. With thorough understanding of the mixed demodulation algorithm, a demodulation system is structured based on this algorithm.

Chapter 4 proposes the hardware implementation of mixed demodulator, and presents the details for quadrature mixer, IIR low pass filter, and DLL individu-

ally. The CORDIC method introduced shows the advantage in get the arctangent value without division operation involvement in the hardware implementation.

Chapter 5 gives detailed test results on signal quality, robustness and processing time and performance analysis is made.

Chapter 6 presents the conclusions with some recommendations for future work.

Chapter 2

A Survey of Digital FM Demodulation System

A typical digital communication system includes analog-to-digital (A/D), coder/ decoder, modulator/demodulator, power amplifier, etc., in which modulation and demodulation are critical parts. There are many modulation techniques available now. Demodulation is reverse process of modulation but more complicated. We shall focus on the FM modulation techniques and its digital full hardware implementation.

2.1 Digital FM Demodulator

The process of varying of a carrier wave in proportion to a modulating signal is known as frequency modulation (FM). The carrier amplitude of a FM wave is kept constant during modulation and so the power associated with a FM wave is constant. During modulation, the carrier frequency increases when the modulating voltage increases positively and it decreases when the modulating voltage becomes negative [6].

From the algorithm's point of view, Baseband delay demodulator, Phase-adapter demodulator and Phase-Locked Loop (PLL) are commonly used. According to

Schnyder and Haller's simulation and comparison results [39], they show that the delay demodulator needs the modulated signal to have a constant amplitude, the phase-adapter demodulator only useful for narrow-band FM, the PLL has worse frequency response to chirp signal.

Dittmer [9], Noguchi, Daido and Nossek [35] summarized the conventional digital modulation techniques and implementation in radio frequency (RF) systems. It gives an overview of the conventional digital modulation techniques and their advantages and disadvantages. Based on that, Sorace [45] studied the details of digital-to-RF conversion vector modulator and Mirabbasi and Martin [34] proposed a spectrally efficient, dc-free modulation scheme called hierarchical quadrature amplitude modulation technique. These achievements represent latest status of the FM modulation/demodulation techniques.

To implement these techniques and practise in the real applications, many new ideas specific to some application field are developed. In 1988, Gibson and McClary [16] implements a MTS second audio program demodulator, which receives 5 successive samples as if they were samples of a sinewave. But this arithmetic needs a division with square of the carrier envelope which puts a round-off error limit to dynamic range. So this solution is applicable to a carrier with a fairly constant amplitude, such as the SAP subcarrier in the composite BTSC signal. In 1990, Konishi, Hitomi, Naka, Oishi and Yamazaki [26] combined the time-base-error corrector and four digital FM demodulators using Digital Signal Processing for HiFi application. But the digital FM demodulator is conventional pulse-count method, which is direct conversion from analog to digital. The resultant signal to noise ratio (SNR) and signal distortion will deteriorate the signal. Shirato and Okada [40] gave a high capacity fully digitalized modulator using parallel phase process techniques special for digital radio system. Song and Lee [44]implemented a FM demodulator fully in software using a quadri-correlator algorithm. Schnyder and Haller implemented the FM demodulator algorithms in DSP processors [39].

Since DSP processors is not really hardware optimized, an interesting point is a

fully digitalized FM demodulator with catering to hardware architecture. Meuth, Schnase and Halling [32] gave an analysis on the possibilities and limitations for a fully digital RF signal synthesis and control. Park, Joe, Choe and Song introduced a fourth order bandpass delta-sigma front end to FM demodulator to achieve high linearity and rejects amplitude-modulation (AM) components. To achieve stable digitalized signal,Ismailogli and Yalcin [21] added a zero-cross detection at IF. But such methods emphasize on the signal processing of front end. Rohkonen, Malo and Kostamovaara [38] implemented a fully integrated FM demodulator using voltage-controlled pulse shrinking delay line interpolator. But such phase line loop is a nonlinear system, which has a characteristic to produce non harmonic frequencies and further causes distortion. From hardware point of view, this solution will consume more power.

To achieve a digital FM demodulator on SOC (system On-Chip), a mixed demodulator algorithm is introduced and implemented on FPGA.

2.2 Trigonometric Functions

During the operation of these demodulation techniques, computation of trigonometric functions is necessary. Look-up table is a normal and effective method. The CORDIC algorithm is an alternative of the solution.

The CORDIC algorithm was first introduced by Volder in [51], as an efficient method to perform plane rotations. This algorithm was later generalized by Walther in [52] for rotations in circular, linear and hyperbolic systems. Duh and Wu [10] implement the discrete Cosine transform using CORDIC techniques. Grayver and Daneshrad [17] introduced another approach to direct digital frequency synthesis based on the CORDIC algorithm. Freeman and O'Donnell [14] designed a versatile signal processor, which can perform multiple rotations, multiplication and additions within one clock cycle using CORDIC rotators. Hekstra and Deprettere [18] developed the floating point CORDIC. The CORDIC algo-

rithm has found its way into diverse applications including the 8087 math coprocessor [12], the HP-35 calculator, radar signal processors [2] and robotics.

CORDIC rotation has also been proposed for computing discrete Fourier, discrete Cosine [7], discrete Hartley [19] and Chirp-Z [20] transforms and filtering, singular value decomposition [41]. According to a survey done by R. Andraka, a so-called unified CORDIC algorithm is developed, which is hardware efficient [1]. Because it is an algorithm and can be implemented by software, it can save cost to the hardware implementation of digital modulator. Vankka, Sanchis, Kosunen and Halonen [50] developed and implemented a CORDIC based QAM modulator using programmable logic devices (PLDs). This becomes a good base to integrate into an application specific integrated circuit (ASIC).

There are a number of ways to implement a CORDIC processor. The ideal architecture depends on the speed versus area tradeoffs in the intended application. An iterative CORDIC architecture can be obtained simply by duplicating each of the three difference equations in hardware as shown in Figure 2.1. The decision function, s_i , is driven by the sign of the Y or Z register depending on whether it is operated in rotation or vectoring mode. In operation, the initial values from the registers are passed through the shifters and adder-subtractors and results placed back in the registers. The shifters are modified on each iteration to cause the desired shift for the iteration. Likewise, the ROM address is incremented on each iteration so that the appropriate elementary angle value is presented to the Z adder-subtractor. On the last iteration, the results are read directly from the adder-subtractors. Obviously, a simple state machine is required keep track of the current iteration and to select the degree of the shift and ROM address for each iteration.

The design depicted in Figure 2.1 uses word-wide data paths (called bit-parallel design).

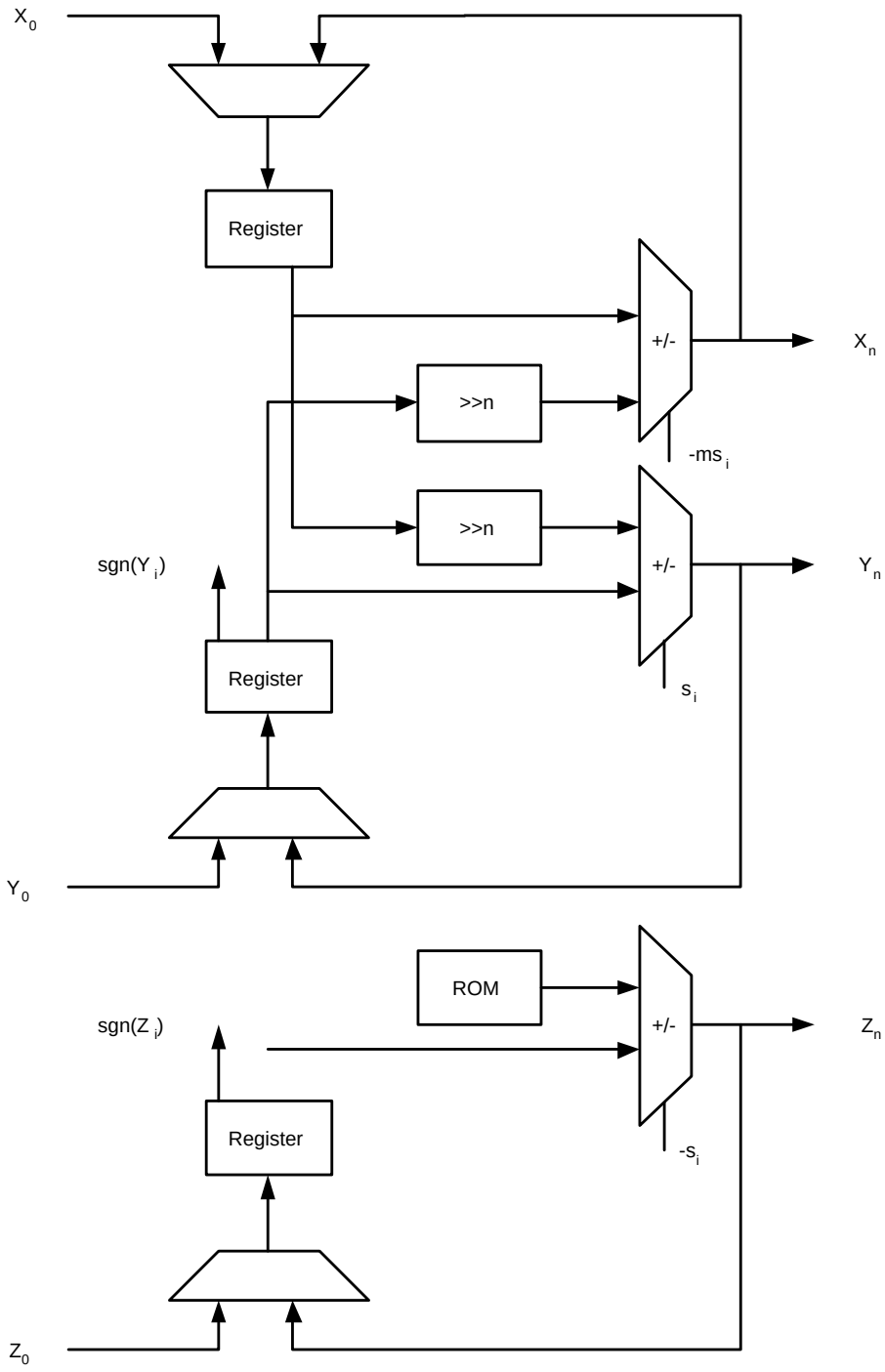

Figure 2.1: Iterative CORDIC structure [1]

A considerably more compact design is possible using bit serial arithmetic. The simplified interconnect and logic in a bit serial design allows it to work at much higher clock rate than the equivalent bit parallel design. Of course, the design also

needs to clock w times for each iteration (w is the width of the data word). The bit serial design consists of three bit serial adder-subtractors, three shift registers and a serial ROM. Each shift register has a length equal to the word width. There is also some gating or multiplexers to select taps off the shift registers for the right shifted cross terms (shift is accomplished using bit delays in bit serial systems). This design requires w clocks for each of the n iterations, where w is precision of the adders. The bit serial design is apparently simpler. The interconnection is minimal and the logic between registers is simple. But the wiring the shift tap multiplexers can present problems in some FPGAs. And the possibility of using extreme bit clock frequencies makes up for the large number of clock cycles required to complete each rotation. The bit serial CORDIC architecture is shown in Figure 2.2.

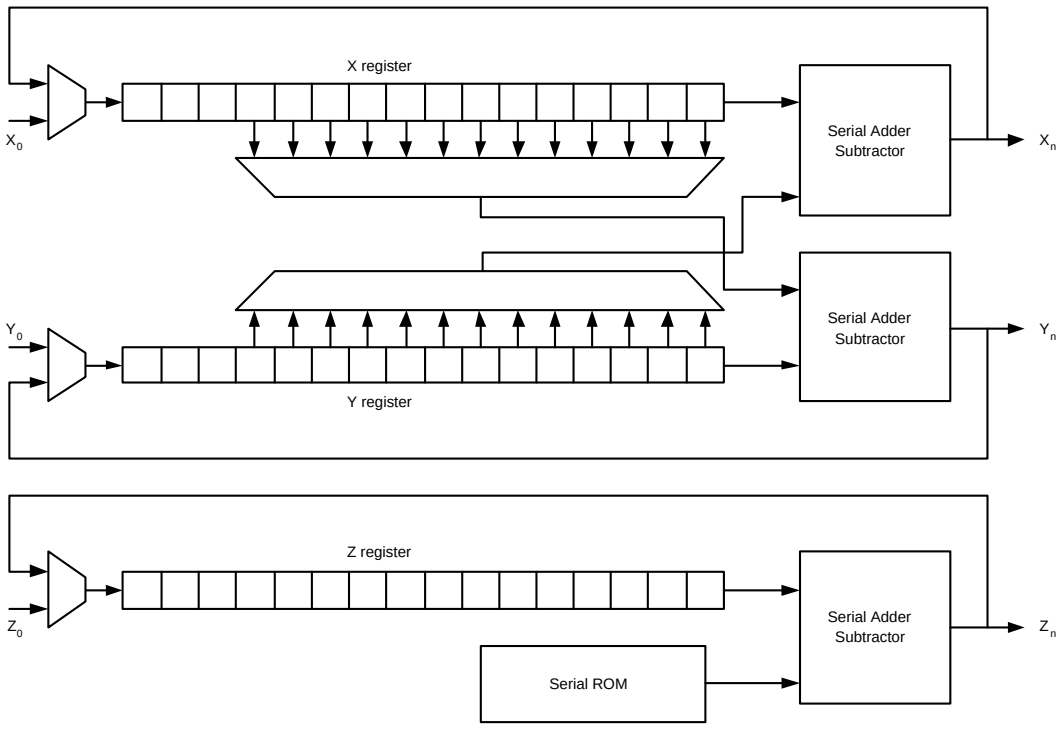

Figure 2.2: Bit serial iterative CORDIC structure [1]

The CORDIC processors discussed above are iterative, which means the processor has to perform iterations at n times the data rate. The iteration process can

be unrolled so that each of n processing elements always performs the same iteration. Unrolling the processor results in two significant simplifications. Firstly, the shifters are each a fixed shift which means that they can be implemented in the wiring. Secondly, the lookup values for the angle accumulator are distributed as constants to each adder in the angle accumulator chain. Those constants can be hardwired instead of requiring storage space. The entire CORDIC processor is reduced to an array of interconnected adder-substractors. The need for registers is eliminated, making the unrolled processor strictly combinatorial. The delay through the resulting circuit would be substantial, but the processing time is reduced from that required by the iterative circuit. The unrolled processor can also be converted to a bit serial design. The structure is shown in Figure 2.3.

Take the example of 16 bit, 8 iteration CORDIC processor, if the design is in Xilinx 4000E series part, the bit serial iterative CORDIC only uses 21 configurable logic blocks (CLBs) while unrolled CORDIC processor occupies 50% of a 4013E and bit parallel iterative CORDIC structure take almost full of resources, as shown in Table 2.1. But the bit serial iterative CORDIC processor is about three and half times slower than bit parallel iterative solution. Even unrolled CORDIC processor takes one and half more process time than bit parallel iterative CORDIC processor.

Table 2.1: Comparison of 16-bit, 8 iterations CORDIC processors on Xilinx 4000E

CORDIC processors	Area	Speed
Bit Serial CORDIC processor	21 CLBs	3.5 times slower than Bit Parallel
Unrolled CORDIC processor	288 CLBs	1.5 times slower than Bit Parallel
Bit Parallel CORDIC processor	570 CLBs	Fastest

Overall of three implementations above, the performance has to be balanced between area and speed. A serial implementation would be smaller but requiring a higher clock frequency for the same speed. The real high performance requires either multiple bit serial processors running in parallel, or an unrolled parallel

pipeline. But such architecture would result in higher area due to the number of required registers and until recently, FPGAs did not have required combination of logic and routing resource to build a high performance parallel processor. The methods discussion here can be used in dedicated ASIC. A good comprise in speed and area is to implementing the simple bit parallel iterative CORDIC processor is used.

2.3 Math Operations

These demodulation techniques also involve a lot of multiplication and divide operations. To map these operations to hardware, more gates are consumed. The most importantly, more resolution to achieve, more clock cycles are needed. To solve this problem, some approaches are in use currently.

One of them is logarithmic number system (LNS). The use of 16-bit LNS arithmetic was originally proposed as an alternative to 16-bit fixed-point by Kingsbury and Rayner in 1971 [25]. Simulated LNS arithmetic demonstrated that the greater dynamic range of this system yielded a very significant improvement in the response of a digital filter. Attention turned some years later to the better round off-error characteristics of the LNS when compared to floating point, and clear improvements in noise-to-signal ratio were demonstrated in a filter [27], and fast Fourier transform (FFT) [47].

Implementation work began with a 1975 paper which suggested a 12-bit device [46], whilst a 1988 proposal extended this to 20-bits [48]. Both designs were direct implementations with a lookup table covering all possible values. It is of course apparent that as the word length increases, the table sizes increase exponentially, which limits the practical utility of this approach to about 20 bits. A 1991 design [54] extended the word length to 28 bits. A separate proposal in 1994 [29]

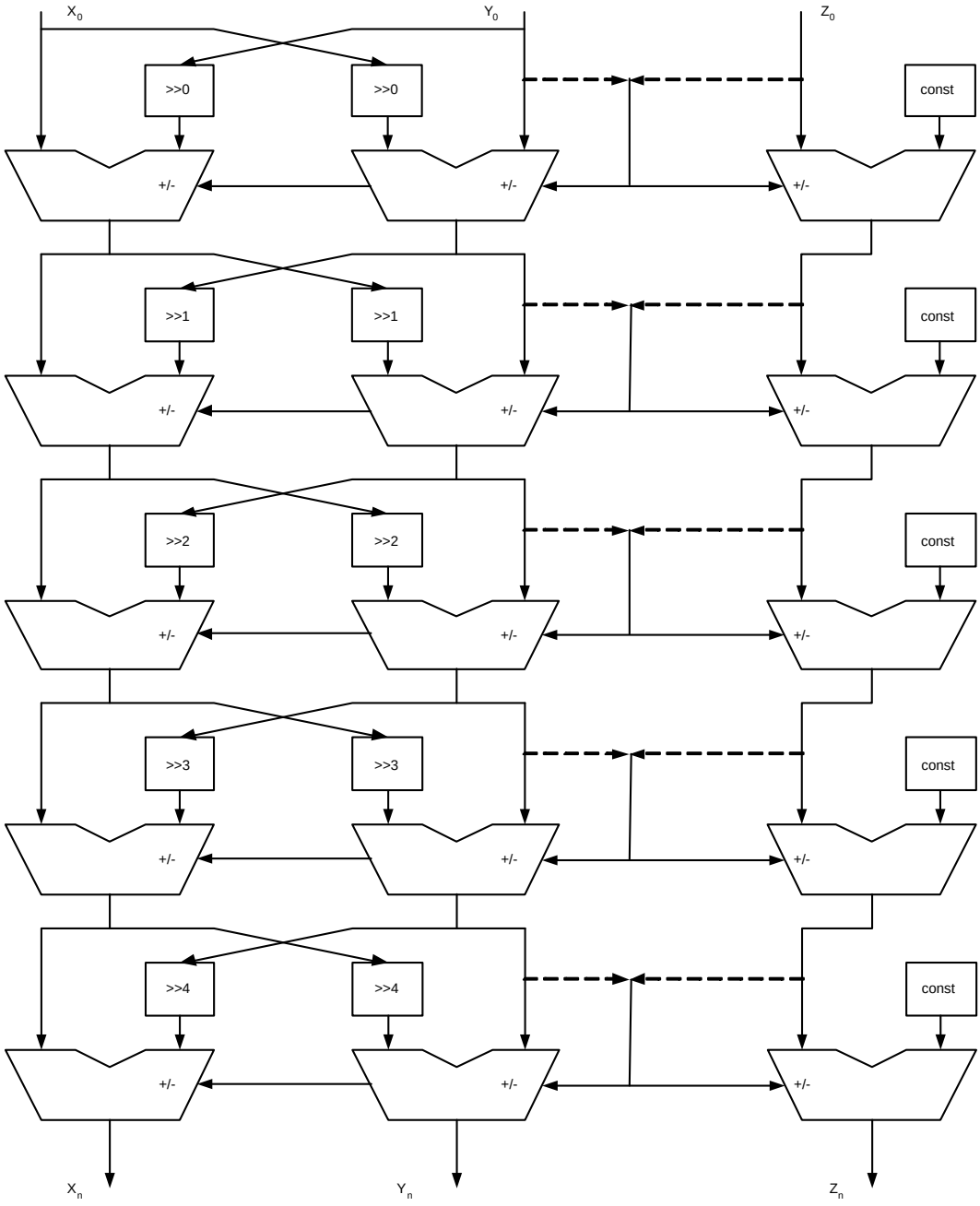

Figure 2.3: Unrolled CORDIC processor [1]

involved the use of a higher-order polynomial interpolator, with a novel scheme for interleaving the stored values so as to reduce the overall memory requirement. A design for a 32-bit unit with a 287 Kbit lookup table was presented, and its addition accuracy shown by simulation to be within float point (FLP) limits.

A variant had a smaller table with larger errors. The critical speed path included a read only memory (ROM), two multipliers, three barrel-shifters, and five carry-propagate adders / subs tractors. Fabrication of the lower-accuracy variant was done in 1995 [30], with a latency of 158 ns in add / subtract operations, and 13 ns for multiply / divide. A 32-bit adder/subs tractor with speed and accuracy comparable to that of a FLP system was developed in 2000 [5] and implemented in FPGA [4].

Some novel arithmetic transforms which may have the potential to simplify the addition and subtraction algebra are given in [49]. This work also includes a review of recent practical applications of LNS arithmetic from the U.S.A. and Japan. Areas cited include image processing and DSP applications, graphics, and aircraft controls. Fujitsu are making some use of the technique in their microprocessors. In this project, luckily the divide operation is avoided via CORDIC algorithms to get the arctangent values.

2.4 Frequency Multiplication

From the survey above, it is obvious that frequency multiplication is required. Phased-locked loops (PLL) and delay locked loop (DLL) are two typically options. Multiple clock outputs can also be de-skewed with respect to one another, to take advantage of multiple clock domains.

The fundamental difference between the PLL and DLL is that instead of a delay line, the PLL uses a voltage controlled oscillator which generates a clock signal that approximates the input clock. The control logic, consisting of a phase detector and filter, adjusts the oscillator frequency and phase to compensate for the clock distribution delay.

Implementation of the DLL or PLL can be accomplished using either analog or digital circuitry; each holds its own advantages. An analog implementation with

careful design can produce a DLL or PLL with a finer timing resolution. Analog implementations can additionally take less silicon area. Conversely, digital implementations offer advantages in noise sensitivity, lower power consumption and jitter performance. Digital implementations also provide the ability to stop the clock, facilitating power management. Analog implementations can require additional power supplies, require close control of the power supply, and pose problems in migration to new process technologies [53]. On the other hand, the oscillator used in the PLL inherently introduces instability and an accumulation of phase error. This in turn degrades the performance of the PLL when attempting to compensate for the delay of the clock distribution network. Conversely, the unconditionally stable DLL architecture does not accumulate phase error. The combination of high operating frequencies and low-power requirements for communication systems today makes clock synthesis and phase synchronization for these devices very challenging. These constraints make all-digital solutions (digital PLLs and DLLs) an attractive option [11, 15, 33].

Delay-locked loops are an attractive alternative to voltage control oscillator (VCO)-based phase-locked loops due to their simpler design and inherent stability [23, 28, 31]. For the digital implementation, the digital delay line architecture is the core. The delay line architecture is comprised of coarse and fine delay elements (CDEs and FDEs). Coarse/fine delay resolution architectures for digital PLLs used in low-power applications and microprocessors [11, 15] and DLLs used in high speed synchronous dynamic random access memory (SDRAM) timing generation [33] have numerous advantages like fast lock-in times, wide operating ranges and low power consumption. However, the drawback is that a consistently smooth transition in delays at CDE boundaries at all pressure volume temperature (PVT) corners is difficult to achieve [11, 15]. Further more, a robust digital delay line architecture is proposed to control the increment/decrement of the fine and coarse delay steps respectively [37].

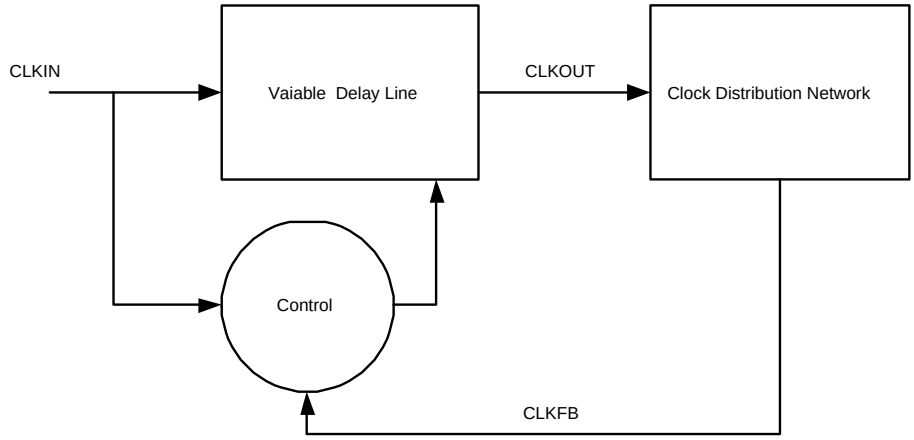

Figure 2.4: Delayed lock loop block diagram

As shown in Figure 2.4, a DLL in its simplest form consists of a variable delay line and control logic. The delay line produces a delayed version of the input clock CLKIN. The clock distribution network routes the clock to all internal registers and to the clock feedback CLKFB pin. The control logic must sample the input clock as well as the feedback clock in order to adjust the delay line.

Delay lines can be built using a voltage controlled delay or as a series of discrete delay elements. Figure 2.5 shows a basic delay element of delay line and the whole delay line is shown in Figure 2.6. A DLL works by inserting delay between the input clock and the feedback clock until the two rising edges align, putting the two clocks 360 degrees out of phase (meaning they are in phase). After the edges from the input clock line up with the edges from the feedback clock, the DLL "locks." As long as the circuit is not evaluated until after the DLL locks, the two clocks have no discernible difference. Thus, the DLL output clock compensates for the delay in the clock distribution network, effectively removing the delay between the source clock and its loads.

In this project, a DLL structure is separately developed but not integrated in since Xilinx FPGAs already integrate the DLL inside. To make a ASIC, a DLL structure is suggested.

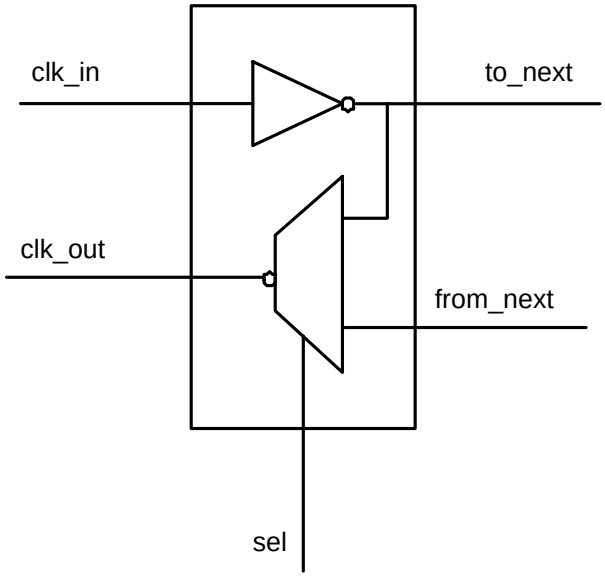

Figure 2.5: Basic delay element

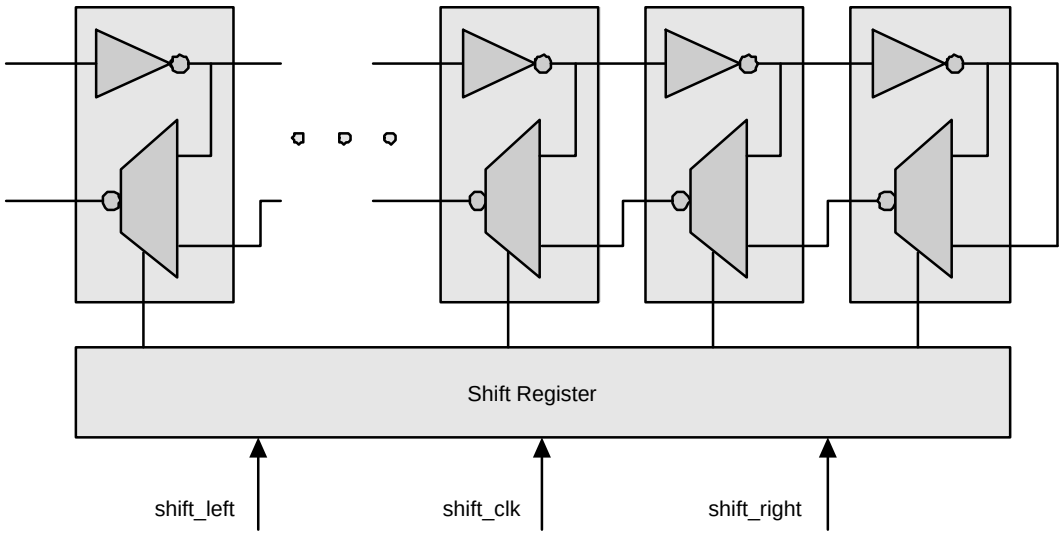

Figure 2.6: Delay line

2.5 Filters and Other Components

Except the modulation techniques themselves, the filters is most critical parts in digital communication systems. Dick and Harris [8] presented several filter designs and the use of CORDIC arithmetic for constructing an FPGA carrier

recovery loop. Pasham, Miller and Chapman introduced and verified constant coefficient multiplier (KCM) concept specific for FPGA implementation of filter algorithms [36].

According to IIR filter equation stated in lecture notes [3], it is easy to draw a structure that implements it. One implementation, called the direct form I structure, is illustrated in Figure 2.7. This structure requires $M+N+1$ multiplications, $M+N$ adds, and $M+N+1$ memory locations to store the input and output vectors. From the Figure 2.7, you can see that the structure can be divided into two sections: one containing an all-zeros filter and one containing an all-pole filter.

Since we are dealing with a linear time invariant (LTI) system, we can switch the order of the two sections and arrive at the direct form II structure shown in Figure 2.8. This structure requires $M+N+1$ multiplications, $M+N$ additions, and $\max M, N$ memory locations. Only the signal $w(n)$ needs to be stored in memory as a vector.

A cascade of lower-order sections can also be used to implement the IIR filters. Each section of the cascade can either be implemented in direct form I or direct form II. Finally, note that both direct form structures are extremely sensitive to quantization the filter coefficients, a_k and b_k.

As mentioned above, digital filter algorithms are primarily composed of multipliers, adders, and registers. The basic structure of an IIR filter is shown in Figure 2.9 based on the direct form II structure. The multipliers and adders form the heart of an IIR filter. The input data passes to the multiplier and then to the adder with interleaving delay elements.

An alternate implementation structure called the transposed form IIR filter is shown in Figure 2.10. Utilizing the same resources, data samples are applied in parallel to all the tap multipliers through pipeline registers. The input registers are not required, because high fan-out input signals can be handled by the FPGA

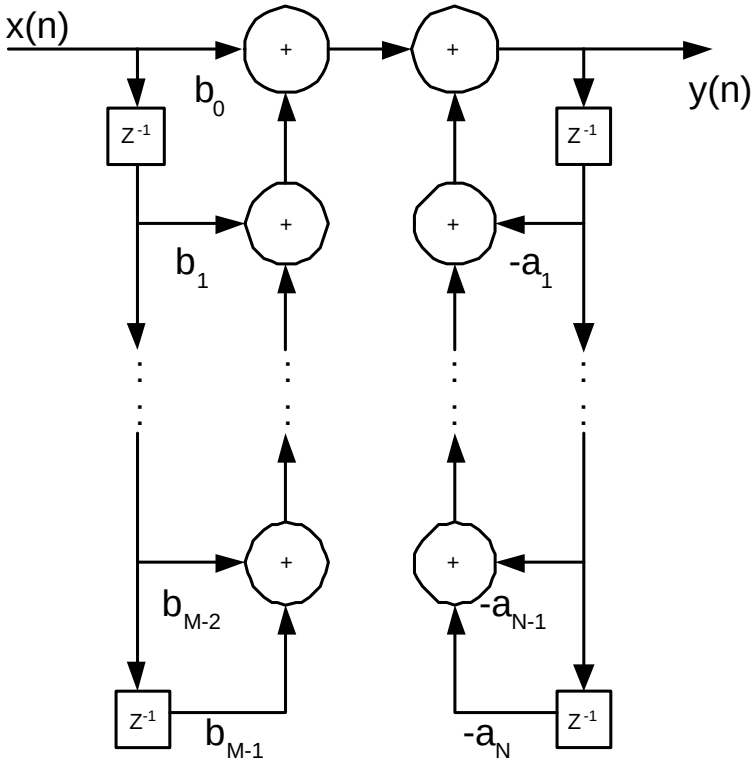

Figure 2.7: Direct form I structure for IIR filters [3]

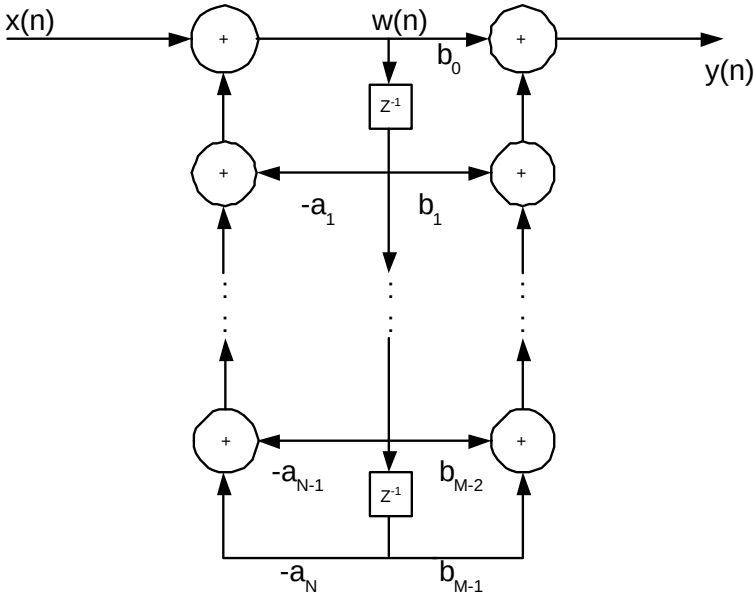

Figure 2.8: Direct form II structure for IIR filters [3]

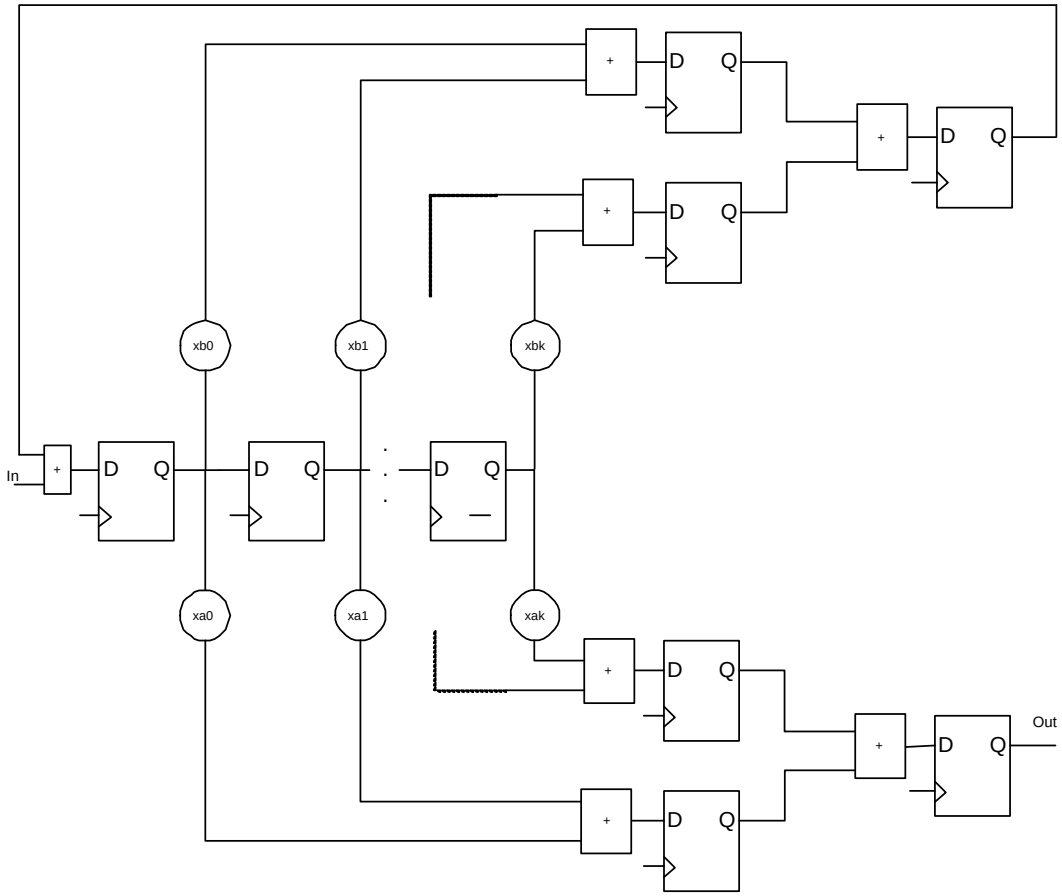

Figure 2.9: IIR filter structure employing tree of pipelined adders

architectures. The products are applied to a cascaded chain of registered adders, combining the effect of accumulators and registers. The order of tap coefficients must be reversed with the first tap closest to the output. This structure allows expansion of the number of taps required in a filter, since each "tap module" is identical. Since the structure is uniform, a single component can be designed and instantiated as many times as required by the number of taps.

Except the modulation techniques and the filters, analog-to-digital converter (ADC) and digital-to-analog converter (DAC) are very important parts in digital communication systems also. Singh [42] implemented a mixed signal, very high speed, analog-to-digital converter. Kester [24] discussed high speed DACs and direct dig-

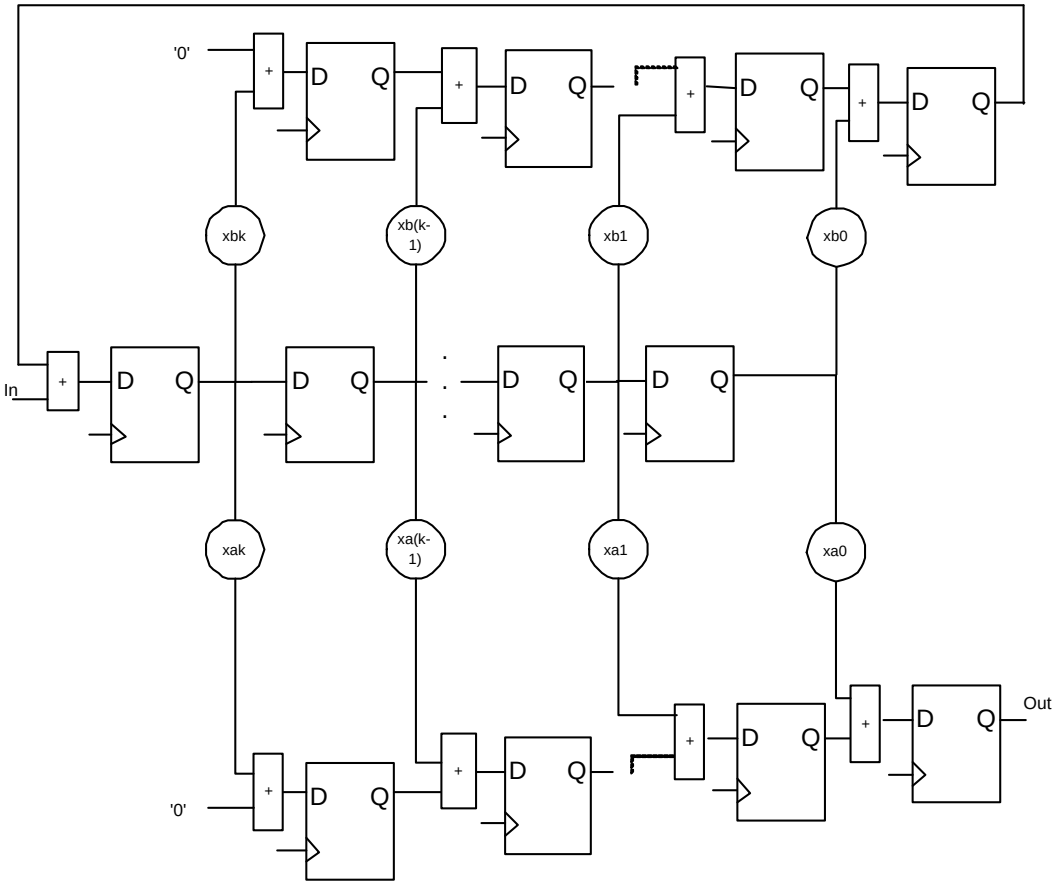

Figure 2.10: Transposed form IIR filters employing cascaded pipelined adders

ital synthesis (DDS) system. Jensen and Galton [22] presented a dynamic element matching technique for low-harmonic distortion digital-to-analog conversion, which can reduce significantly hardware complexity with no reduction in performance.

Chapter 3

Digital Demodulation of FM Signals

3.1 Frequency Modulation

The process of varying the frequency of a carrier wave in proportion to a modulating signal is known as frequency modulation (FM). The carrier amplitude of an FM wave is kept constant during modulation and so the power associated with an FM wave is constant. During modulation, the carrier frequency increases when the modulating voltage increases positively and it decreases when the modulating voltage becomes negative. Frequency modulation (FM) is a type of angle-modulated signal, which has the general form given by (3.1):

$$v_c = V_c \sin\left(\omega_c t - m_f \cos \omega_m t\right) \tag{3.1}$$

where ω_c is the carrier frequency. Let the instantaneous frequency at any time t be f_i. Then (3.1) can be expressed as

$$v_c = V_c \sin \omega_i t \tag{3.2}$$

Relating ω_i to the expression in (3.1), we have

$$\omega_i = \omega_c + \Delta\omega_c \sin\left(\omega_m t\right) \tag{3.3}$$

where $\Delta\omega_c$ is the frequency deviation due to the modulating signal of frequency f_m.

The instantaneous frequency is the time derivative of the instantaneous carrier phase ϕ_i.

$$d\phi_i/dt = \omega_i = \omega_c + \Delta\omega_c \sin(\omega_m t) \tag{3.4}$$

Therefore the instantaneous carrier phase can be expressed as

$$\phi_i = \omega_c t - m_f \cos(\omega_m t) \tag{3.5}$$

where

$$m_f = \frac{\Delta\omega_c}{\omega_m}$$

Expanding v_c yields

$$v_c = V_c \left[\sin\omega_c t \cos((m_f cos\omega_m t)) - cos(\omega_c t sin(m_f cos\omega_m t))\right] \tag{3.6}$$

Now,

$$\begin{aligned}
\cos(m_f \cos\omega_m t) &= J_0(m_f) - 2J_2(m_f) cos 2\omega_m t + 2J_4(m_f)\cos 4\omega_m t - ... \\
\sin(m_f \cos\omega_m t) &= 2J_1(m_f)\cos\omega_m t - 2J_3(m_f)\cos 3\omega_m t + ...
\end{aligned}$$

The coefficients $J_n(m_f)$ are order n Bessel functions of the first kind.

Substituting into v_c yields the result

$$\begin{aligned}
v_c &= V_c[J_0(m_f)\sin\omega_c t - J_1(m_f)\{\cos(\omega_c + \omega_m t\} \\
&= J_2(m_f)\{\sin(\omega_c + 2\omega_m)t + sin\omega_c - 2\omega_m)t\} + ...]
\end{aligned}$$

which reveals an infinite set of sidebands whose amplitudes are determined by the Bessel functions $J_0(m_f)$, $J_1(m_f)$, etc. For small values of m_f, there are few sideband frequencies with large amplitudes. If m_f is large, there are many sideband frequencies but with smaller amplitudes. Hence, it is only necessary to consider a finite number of significant sideband components.

In general, FM signals are not bandlimited. However, their spectra decline quickly outside a bandwidth around the carrier frequency. This bandwidth can be found

using Carson's rule:

$$b_{FM} = 2(\Delta F + f_g) \tag{3.7}$$

where f_g is the highest frequency of the baseband signal. Hence the bandwidth rises with a rising ΔF and a rising message frequency.

3.2 Demodulation Algorithm

Since the digital FM demodulation algorithm needs the FM signal in the baseband, the FM demodulator can be divided into three subsystems: the subsampling unit, the quadrature mixer and the baseband FM demodulator. Figure 3.1) shows this division in block diagram form.

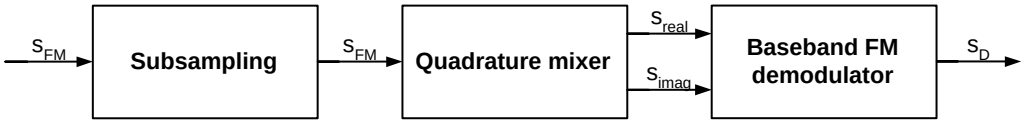

Figure 3.1: Subdivisions of FM demodulation block

Normally, the carrier frequency is more than ten mega hertz level. The sampling rate can not reached so high in analog to digital converters nowadays. In this project, the FM signal after the bandpass filter has a carrier frequency of 10.7 MHz and a bandwidth of 12.5 kHz. This results in a maximum frequency of over 10.7 MHz. Hence, a sampling rate of over 21 MHz is required. This data rate is too fast for today devices. However as the signal is frequency limited (Bandwidth b), a subsampling is possible and the sampling rate can be calculated as follows:

In the special case that

$$f_1 = \lambda b, \quad \lambda \in [\mathbb{N}] \tag{3.8}$$

$$f_2 = (\lambda + 1)b \tag{3.9}$$

the sample rate is

$$f_A = 2b \tag{3.10}$$

for a non aliasing periodic sequel of the spectrum. The subsampling can expressed in terms of the carrier frequency f_c. In this case,

$$f_c - \frac{b}{2} = \lambda b \tag{3.11}$$

$$f_c = b\left(\frac{2\lambda+1}{2}\right) \tag{3.12}$$

For a general carrier frequency f_c and bandwidth b, the condition of an even λ is often not fulfilled. Thus, the bandwidth has to increase.

$$b' = bq,\ q > 1 \tag{3.13}$$

The new bandwidth is therefore

$$f_T = b'\left(\frac{2\lambda+1}{2}\right) \tag{3.14}$$

$$b' = \frac{2f_T}{2\lambda+1} \tag{3.15}$$

here λ is a largest integer that is smaller than $(f_T - \frac{b}{2})/b$. Therefore the sampling rate is

$$f_A = 2b' = \frac{4f_T}{2\lambda+1} \tag{3.16}$$

The mixing to the baseband is carried out by the multiplication of the FM signal and a complex oscillator $e^{j\omega_T n}$ and a low pass filter as shown in Figure 3.2.

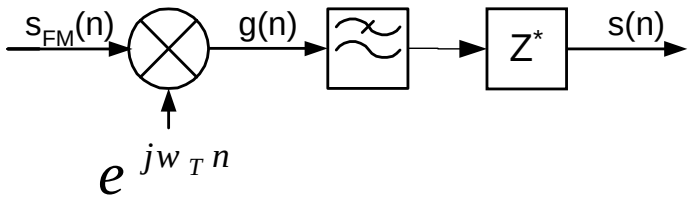

Figure 3.2: Quadrature-mixer

The input signal is the modulated signal s_{FM}

$$s_{FM}(n) = A\cos\left[\omega_T n + \phi_{FM}(n)\right] \tag{3.17}$$

The output signal of the mixer is:

$$\begin{aligned} g(n) &= s_{FM}(n)e^{j\omega_T n} \\ &= A\cos(\omega_T n + \phi_{FM}(n))e^{j\omega_T n} \\ &= A\frac{e^{j(\omega_T n+\phi_{FM}(n))} + e^{-j(\omega_T n+\phi_{FM}(n))}}{2}e^{j\omega_T n} \\ &= \frac{A}{2}[e^{j(\omega_T n+\phi_{FM}(n)+\omega_T n)} + e^{j(-\omega_T n-\phi_{FM}(n)+\omega_T n)}] \\ &= \frac{A}{2}[e^{j(2\omega_T n+\phi_{FM}(n))} + e^{j(-\phi_{FM}(n))}] \end{aligned}$$

$$\begin{aligned} s(n) &= (g(n) * TP)^* = (\frac{A}{2}e^{-j\phi_{FM}(n)})^* \\ &= \frac{A}{2}e^{j\phi_{FM}(n)} = \frac{A}{2}cos(\phi_{FM}(n)) + j\frac{A}{2}sin(\phi_{FM}(n)) \end{aligned} \tag{3.18}$$

The result is a complex signal s. The mixer can also be realized with real signals by multiplying the FM signal with a sine and cosine oscillation signal as shown in Figure 3.3.

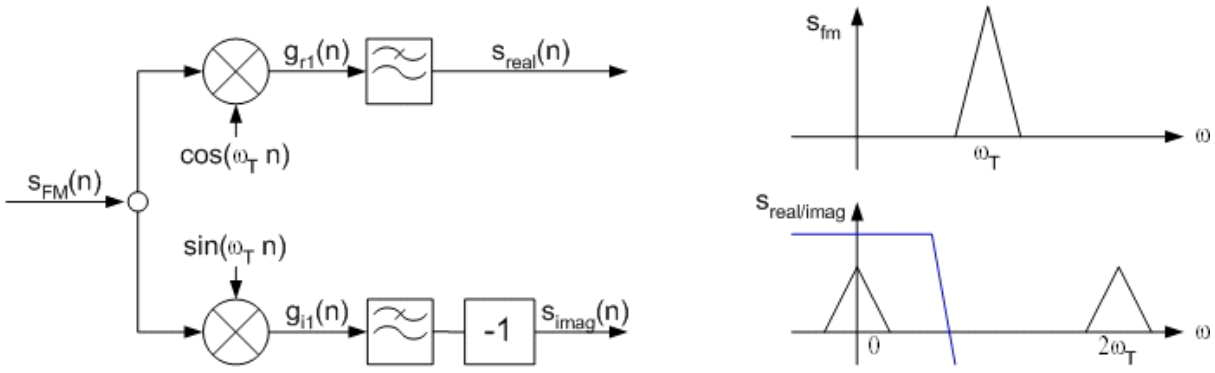

Figure 3.3: Real quadrature-mixer [39]

The input signal is again the FM signal s_{FM}.

$$s_{FM}(n) = A\cos(\omega_T n + \phi_{FM}(n))$$

Therefore, the output signals are

$$\begin{aligned} g_{r1}(n) &= s_{FM}(n)\underline{\cos(\omega_T n)} \\ &= A\cos(\omega_T n + \phi_{FM}(n))\cos(\omega_T n) \\ &= \frac{A}{2}\left[\cos(\omega_T n + \phi_{FM}(n) - \omega_T n) + \cos(\omega_T n + \phi_{FM}(n) + \omega_T n)\right] \\ &= \underline{\frac{A}{2}\cos(\phi_{FM}(n))} + \frac{A}{2}\cos(2\omega_T n + \phi_{FM}(n)) \end{aligned}$$

$$s_{real}(n) = g_{r1}(n) * g_{TP}(n) = \frac{A}{2}\cos(\phi_{FM}(n)) \tag{3.19}$$

$$\begin{aligned} g_{i1}(n) &= s_{FM}(n)\underline{\sin(\omega_T n)} \\ &= A\cos(\omega_T n + \phi_{FM}(n))\sin(\omega_T n) \\ &= \frac{A}{2}\left[\sin(-\omega_T n - \phi_{FM}(n) + \omega_T n) + \sin(\omega_T n + \phi_{FM}(n) + \omega_T n)\right] \\ &= \underline{\frac{A}{2}\sin(-\phi_{FM}(n))} + \frac{A}{2}\sin(2\omega_T n + \phi_{FM}(n)) \end{aligned}$$

$$s_{imag}(n) = (-1)\, g_{i1}(n) * g_{TP}(n) = \frac{A}{2}\sin(\phi_{FM}(n)) \tag{3.20}$$

It results in two signal, the real part s_{real} and the imaginary part s_{imag}, also known as the **I** (Inphase) and **Q** (Quadraturephase) signals.

3.2.1 Demodulator Algorithm Details

Figure 3.4 shows the block diagram of a demodulator. Since this demodulator algorithm is a combination of the delay demodulator and the phase adapter Demodulator, it is called mixed demodulator, which is referred from Schnyder and Hall's report [39]. In this way, some of the disadvantages can be removed.

The input signals are the two real signals s_{real} and s_{imag} after the mixing (3.19)

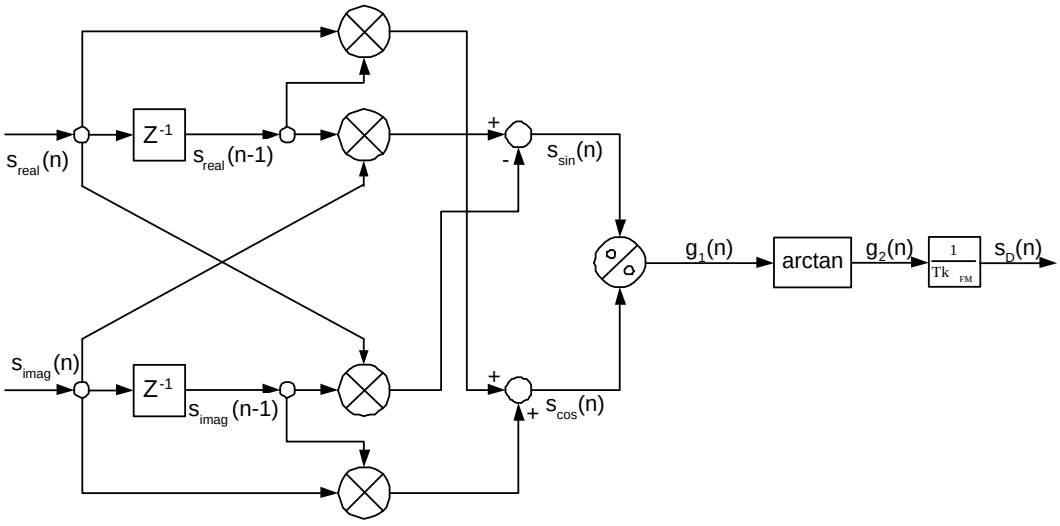

Figure 3.4: Mixed demodulator

and (3.20). From (3.2) the output signal can be written as:

$$\begin{aligned} s_{\sin}(n) &= s_{imag}(n)\, s_{real}(n-1) - s_{real}(n)\, s_{imag}(n-1) \\ &= \sin(\phi_{FM}(n)) \cos(\phi_{FM}(n-1)) - \cos(\phi_{FM}(n)) \sin(\phi_{FM}(n-1)) \\ &= \sin(\phi_{FM}(n) - \phi_{FM}(n-1)) \end{aligned}$$

$$\begin{aligned} s_{\cos}(n) &= s_{real}(n)\, s_{real}(n-1) + s_{imag}(n)\, s_{imag}(n-1) \\ &= \cos(\phi_{FM}(n)) \cos(\phi_{FM}(n-1)) + \sin(\phi_{FM}(n)) \sin(\phi_{FM}(n-1)) \\ &= \cos(\phi_{FM}(n) - \phi_{FM}(n-1)) \end{aligned}$$

$$\begin{aligned} g_1(n) &= \frac{s_{\sin}(n)}{s_{\cos}(n)} = \frac{\sin(\phi_{FM}(n) - \phi_{FM}(n-1))}{\cos(\phi_{FM}(n) - \phi_{FM}(n-1))} = \tan(\phi_{FM}(n) - \phi_{FM}(n-1)) \\ g_2(n) &= \arctan(g_1(n)) = \phi_{FM}(n) - \phi_{FM}(n-1) \end{aligned}$$

$$s_D(n) = \frac{g_2(n)}{Tk_{FM}} = \frac{\phi_{FM}(n) - \phi_{FM}(n-1)}{Tk_{FM}} = \frac{\phi'_{FM}(n)}{k_{FM}} = s_N(n) \qquad (3.21)$$

The mixed demodulator demodulates the FM signals in the baseband.

The signal after the arctangent $g_2(n)$ must be limited between $-\frac{\pi}{2}$ and $\frac{\pi}{2}$. The maximal frequency deviation can be calculated for flawless demodulation.

$$k_{FM}\hat{s}_N = \Delta F 2\pi \tag{3.22}$$

$$\frac{2\pi\Delta F}{f_A} < \frac{\pi}{2} \tag{3.23}$$

$$\Delta F < \left(\frac{\pi}{2}\right)\left(\frac{f_A}{2\pi}\right) < \frac{f_A}{4} \tag{3.24}$$

It shows that the maximal ΔF depends on the sampling rate. Therefore it is not limited, because with rise of ΔF also the bandwidth of the FM-Signal rises. Thus the sample rate has to be increased.

Because of the limitation of the signal $g_2(n)$ the problem of a division by zero does not exist. The signal

$$s_{\cos}(n) = \cos\left(\phi_{FM}(n) - \phi_{FM}(n-1)\right) = \cos\left(g_2(n)\right) \tag{3.25}$$

can be zero only when $\phi_{FM} = \pm\frac{\pi}{2}i$, where i is an odd integer. To avoid a division by zero, the output of the delay z^{-1} for the first sample needs to be initialized to a non-zero number.

The received FM signal due to distortion in the channel is not known. However the mixed demodulator needs a constant amplitude which is achieved by normalizing the magnitude of the signal. This is done by dividing the real signals. The output signals are

$$\begin{aligned} out(n) &= \frac{s_{real} + j s_{imag}}{|s_{real} + j s_{imag}|} \\ &= \frac{s_{real} + j s_{imag}}{\sqrt{s_{real}^2 + s_{imag}^2}} \\ &= \frac{s_{real}}{\sqrt{s_{real}^2 + s_{imag}^2}} + j\frac{s_{imag}}{\sqrt{s_{real}^2 + s_{imag}^2}} \end{aligned}$$

The result is a constant amplitude signal normalized to one.

3.2.2 Trigonometric Calculation Using CORDIC Algorithms

CORDIC is an iterative algorithm for computing trigonometric, hyperbolic and transcendental functions in a compute efficient manner. Volder [51] first explained CORDIC algorithm in 1959 for real time air-borne applications. In CORDIC, the functions are computed by simple shift and add operations. To explain the basic concept of CORDIC, consider a two-dimensional Euclidean space as shown in Figure 3.5.

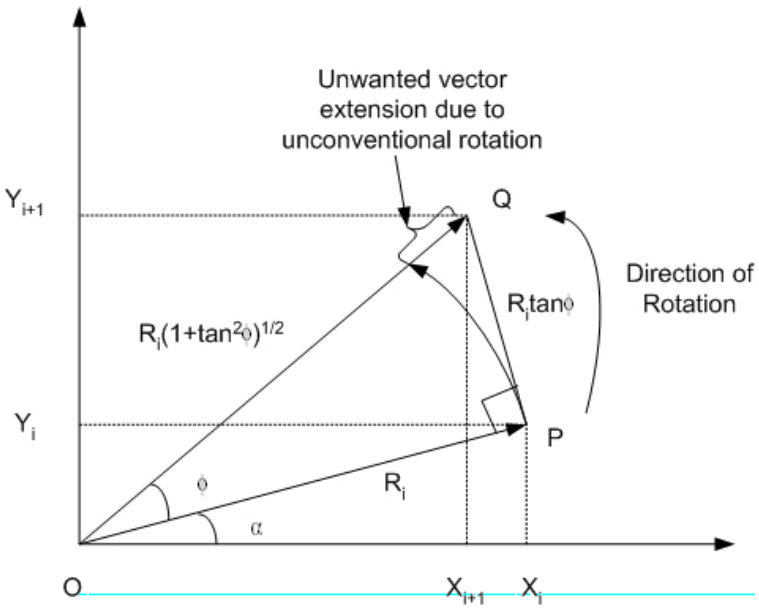

Figure 3.5: Vector rotations

Let X_i and Y_i be the x and y coordinates of the vector **OP** with magnitude R_i. This vector is rotated through an angle f to form the new vector **OQ**. The rotation is not a pure vector rotation but a motion of vector **OP** along the tangent of the circle formed by **OP** as radius at the point P.

Then the resultant vector will have a magnitude given by $R_i(1 + tan^2\phi)^{\frac{1}{2}}$ which is $R_i sec\phi$. The coordinates of **OQ** are derived as follows:

$$X_i - X_{i+1} = R_i cos\alpha - R_i sec\phi cos(\phi + \alpha) \tag{3.26}$$

Expanding we get

$$X_i - X_{i+1} = R_i cos\alpha - R_i sec\phi(cos\phi cos\alpha - sin\phi sin\alpha) \tag{3.27}$$

$$X_i - X_{i+1} = R_i cos\alpha - R_i cos\alpha + R_i sin\alpha tan\phi \tag{3.28}$$

$$X_i - X_{i+1} = R_i sin\alpha tan\phi$$

$$X_i - X_{i+1} = Y_i tan\phi$$

$$X_{i+1} = X_i - Y_i tan\phi \tag{3.29}$$

Similarly

$$Y_{i+1} = Y_i + X_i tan\phi \tag{3.30}$$

Taking into consideration the direction of rotation (s), s = '+1' for anticlockwise rotation and s = '-1' for clockwise rotation. The Eq. 3.29 and 3.30 can be expressed as

$$X_{i+1} = X_i - s_i Y_i tan\phi \tag{3.31}$$

$$Y_{i+1} = Y_i + s_i X_i tan\phi \tag{3.32}$$

In CORDIC algorithm the angle of rotation is achieved by a series of micro-rotations. In simple words the input angle is decomposed into small micro-angles that take values of $\arctan 2^{-i}$, where I take values from 0 to N. This decomposition of the input angle is based on the fact that 2^{-i} is approximately equal to $\arctan 2^{-i}$. The input angle, which is represented in radians, is split into its respective negative powers of 2 or in other words the input angle is represented as the sum of the values of the binary bit positions. Thus the rotation through a given angle is carried out

by a series of micro-rotations, each rotation corresponding to $\arctan 2^{-i}$ which is governed by the bits of the input angle. Figure 3.6 depicts the micro-rotations.

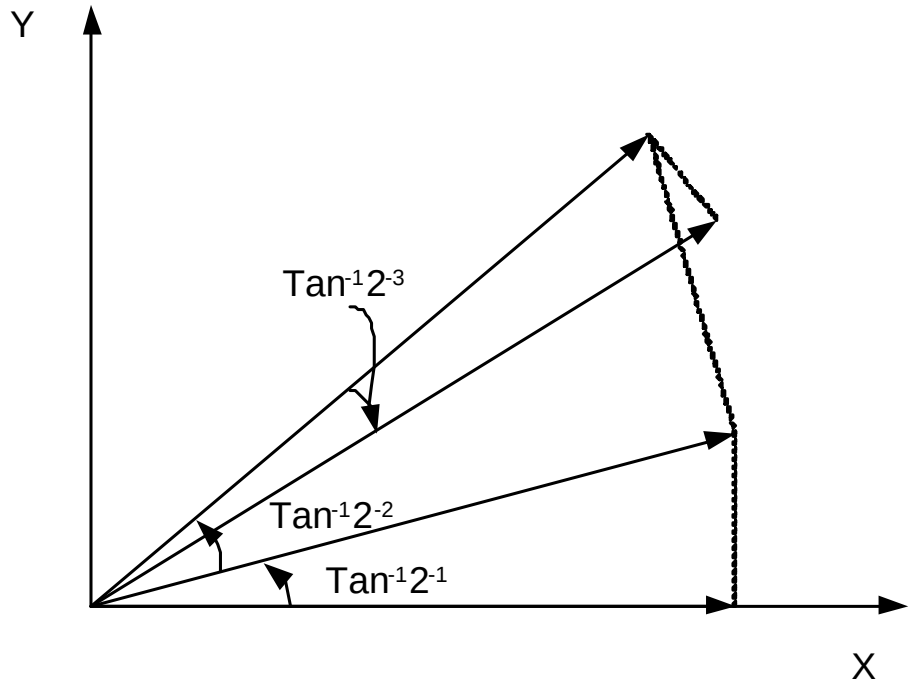

Figure 3.6: CORDIC micro-rotations

So the multiplication by the tangent term is reduced to simple shift operation. Arbitrary angles of rotation are obtainable by performing a series of successively smaller elementary rotations. The decision at each iteration, i, is which direction to rotate rather than whether or not to rotate.

The angle of a composite rotation is uniquely defined by the sequence of the directions of the elementary rotations. That sequence can be represented by a decision vector. The set of all possible decision vectors is an angular measurement system based on binary arctangent. The angle accumulator adds a third difference equation to the CORDIC algorithm:

$$Z_{i+1} = Z_i - s_i tan^{-1}(2^{-1}) \tag{3.33}$$

Thus, the CORDIC algorithm can be described best as follows:

Input X coordinate: X_0 Input Y coordinate: Y_0

Input Z (angle):θ

for i = 0 to N (desired accuracy of operation in bits) loop

$X_{i+1} = X_i - s_i Y_i tan(tan^{-1}2^{-i}) = X_i - s_i Y_i 2^{-i}$

$Y_{i+1} = X_i + s_i Y_i 2^{-i}$

$Z_{i+1} = Z_i - s_i \arctan 2^{-i}$

End loop

where $s_i = +1$ *when* $Z_{i+1} \geq 0.0$, $s_i = -1$ *else*

In Figure 3.6, $X_0 = 1$ and $Y_0 = 0$. So after N micro-rotations the final X coordinate, X_N should be $cos\theta$ and final Y coordinate, Y_N should be $sin\theta$. But that is not the case. Due to unconventional vector rotation it is evident that the resultant vector after each micro-rotation is amplified in magnitude (Figure 3.5). This inherent amplification adds on to the vector in the subsequent micro-rotations. This amplification factor is termed as the K-factor and has to be compensated as the end to obtain the correct values of sine and cosine. From Figure 3.5, the K-factor for the i^{th} iteration or micro-rotation is given by

$$k_i = \sqrt{(1 + s_i^2 2^{-2i})} \tag{3.34}$$

The product of the k_i's can be applied elsewhere in the system or treat as part of a system processing gain. The product approaches 0.6073 as the number of iterations goes to infinity. Therefore, the rotation algorithm has a gain, K of approximately 1.647. the exact gain depends on the number of iterations and obeys the relation

$$K = \prod_{i=0}^{N-1} (\sqrt{(1 + s_i^2 2^{-2i})}) \tag{3.35}$$

To obtain the correct values of sine and cosine,

$$sin\theta = X_N / K$$

$$cos\theta = Y_N/K$$

If $X_0 = 1$ *and* $Y_0 = b$ *and* $Z_0 = \theta$, then

$$X_N = K(acos\theta - bsin\theta) \tag{3.36}$$

$$Y_N = K(bcos\theta - asin\theta) \tag{3.37}$$

This mode of CORDIC, where a vector is rotated over a given input angle, is called the rotation mode. The input coordinate is rotated until the angle left for rotation tends to zero. The final coordinates of X and Y after vector compression gives the cosine and sine of the input angle respectively (if the starting vector has unit magnitude and lies on the X-axis). When the reverse occurs it is called the Vectoring mode. In vectoring mode the input vector is rotated until the Y coordinate tends to zero. Then the X coordinates corresponds to the magnitude of the input vector (with vector compression) and the angle gives the argument of the input vector. This implies a Cartesian to polar conversion. The CORDIC equations for vectoring mode are given by:

Input X coordinate: X_0

Input Y coordinate: Y_0

Input Z (angle): 0

for i = 0 to N (desired accuracy of operation in bits) loop

$$X_{i+1} = X_i + s_i Y_i \tag{3.38}$$

$$Y_{i+1} = Y_i - s_i X_i 2^{-i} \tag{3.39}$$

$$Z_{i+1} = Z_i + s_i \arctan 2^{-i} \tag{3.40}$$

End loop

Where $s_i = +1$ when $Y_{i+1} \geq 0.0 = -1$ *else*

The K-factor remains the same. The final X coordinate corresponds to $K(X_0^2 + Y_0^2)^{\frac{1}{2}}$ and final angle corresponds to $tan^{-1}(\frac{Y_0}{X_0})$.

Sine and Cosine

The rotational mode CORDIC operation can simultaneously compute the sine and cosine of the input stage. Setting the Y component of the input vector to zero reduces the rotation mode result to:

$$X_n = K_n X_0 cos Z_0 \tag{3.41}$$

$$Y_n = K_n X_0 sin Z_0 \tag{3.42}$$

By setting X_0 equal to $1/K_n$, the rotation produces the unscaled sine and cosine of the angle argument, Z_0. Very often, the sine and cosine values modulate a magnitude value. Using other techniques (e.g., a look up table) requires a pair of multipliers to obtain the modulation. The CORDIC technique performs the multiply as part of the rotation operation, and therefore eliminates the need for a pair of explicit multipliers. The output of the CORDIC rotator is scaled by the rotator gain. If the gain is not acceptable, a single multiply by the reciprocal of the gain constant placed before the CORDIC rotator will yield unscaled results. It is worth noting that the hardware complexity of the CORDIC rotator is approximately equivalent to that of a single multiplier with the same word size.

Arctangent

The arctangent is directly computed using the vectoring mode CORDIC rotator if the angle accumulator is initialized with zero. The argument must be provided as a ratio expressed as a vector (x,y). Presenting the argument as a ratio has the advantage of being able to represent infinity (by setting x=0). Since the arctangent result is taken from the angle accumulator, the CORDIC rotator growth does not

affect the result.

$$Z_n = Z_0 + tan^{-1}(\frac{Y_0}{X_0}) \tag{3.43}$$

Vector magnitude

The vectoring mode CORDIC rotator produces the magnitude of the input vector as a byproduct of computing the arctangent. After the vectoring mode rotation, the vector is aligned with the x axis. The magnitude of the vector is therefore the same as the x component of the rotated vector. This result is apparent in the result equations for the vector mode rotator:

$$X_n = K_n\sqrt{X_0^2 + Y_0^2} \tag{3.44}$$

The magnitude result is scaled by the processor gain, which needs to be accounted for elsewhere in the system. This implementation of vector magnitude has a hardware complexity of roughly one multiplier of the same width. The CORDIC implementation represents a significant hardware savings over an equivalent Pythagorean processor. The accuracy of the magnitude result improves by 2 bits for each iteration performed.

3.3 Summary

In this chapter, the algorithm for digital FM demodulation system is presented. To solve the trigonometric calculation, CORDIC is introduced. The mixed demodulator algorithm is selected to implemented in FPGA/ASIC in this work due to its good signal quality and robust performance.

Chapter 4

Hardware Architecture And Implementation

In this chapter, the implementation of the mixed demodulation algorithm discussed in Chapter 3 is presented. Figure 4.1 shows the system architecture of the mixed demodulator. The three main parts are the quadrature mixer, mixed demodulator algorithm and IIR filters. Details of the implementation of these sub-systems are given in the following sections.

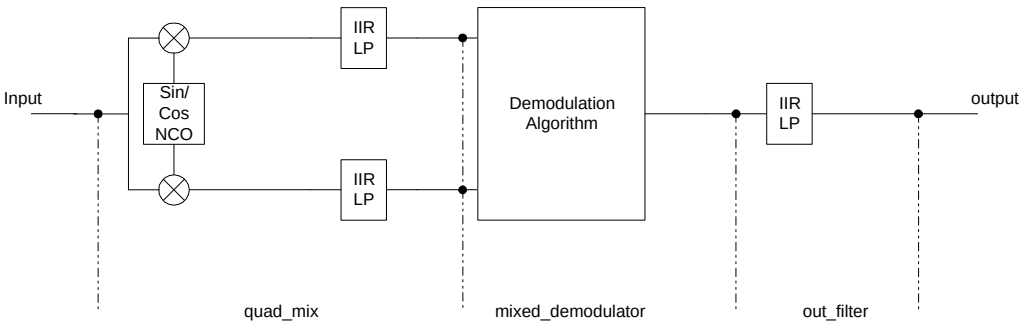

Figure 4.1: System architecture

4.1 Quadrature Mixer

Figure 4.2 shows the block diagram of the quadrature mixer. It involves two modules: the sine (cosine) oscillator and the low-pass filter. The oscillators generate sine (cosine) signals at the angular frequency ω_M. The filters operate at sample intervals of T seconds.

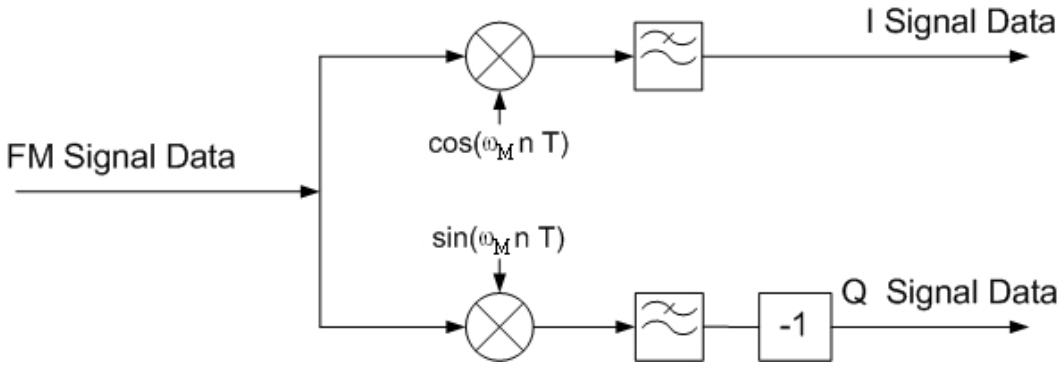

Figure 4.2: Block diagram of quadrature mixer

4.1.1 Sine and Cosine Oscillators

The most straightforward way to calculate the sine or cosine is to use a look-up table in most hardware implementation. Recently, CORDIC algorithm explored in Section **??** is also used widely. But the nature of the CORDIC algorithm is iterative. It means the resolution will really depends on the number of iteration. More iterations it runs, more resolution the result can be achieved. To map to the hardware implementation, it means more clock cycles are consumed.

For calculation in quadrature mixer, the period of sine/cosine is decided by modulation frequency f_M, the step of calculation is based on the sample time T. Once an input data is sampled in, the results must be available. Since the mixer is running at sampling frequency, lookup table method is preferred to get the sine/cosine result to match the timing rather than using CORDIC algorithm.

Different structures may be theoretically equivalent, but behave differently when implemented in finite precision. Since the result of sine/cosine is between -1 and 1, the data in look-up table is expressed in Q15 format.

Q15 means that a 16-bit word is used to express a signed number between positive and negative one. The most-significant binary digit is interpreted as the sign bit in any Q format number. Thus, in Q15 format, the decimal point is placed immediately to the right of the sign bit. The fractional portion to the right of the sign bit is stored in regular two's complement format as shown in Figure 4.3.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value	S	Q 14	Q 13	Q 12	Q 11	Q 10	Q 9	Q 8	Q 7	Q 6	Q 5	Q 4	Q 3	Q 2	Q 1	Q 0

Figure 4.3: Q15 bit fields.

The approximate allowable range of numbers in Q15 representation is (-1,1) and the finest fractional resolution is $2^{-15} = 3.05 \times 10^{-5}$. The integer value of an Q15 value can be computed using equation:

$$i = round(f \times 2^{15}) \tag{4.1}$$

where i is the integer value and f is the fractional value. For example: $i = round(0.5 \times 32768) = 16384$.

The following subsection shows how to perform the product operation on two fixed point numbers, $a = n2^{-p}$ and $b = m2^{-q}$, expressing the answer in the form $c = k2^{-r}$, where p, q and r are fixed constant exponents.

The product $c = a \times b$ can be performed using a single integer multiplication. From the equation:

$$a \times b = n2^{-p} \times m2^{-q} = (n \times m)^{(p+q)} \tag{4.2}$$

It follows that the product $n \times m$ is the mantissa of the answer with exponent $p+q$. To convert the answer to have exponent r, perform shifts as described above. For

example, two Q15 numbers:

$$k = (n \times m) \gg 15 \tag{4.3}$$

For 16-bit data format, the multiplication $Q15 \times Q15$ gives Q30 result as shown in Figure 4.4. The result is a number with 2 sign bits and 30 fractional bits. The sign bits always happen to be the same, so that we can remove the redundancy.

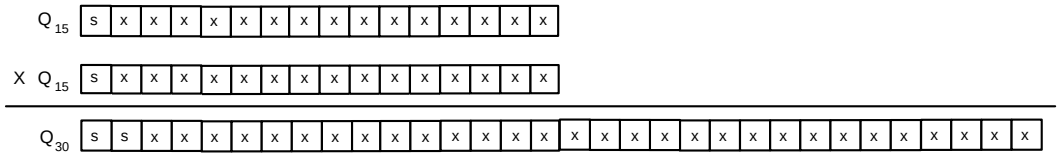

Figure 4.4: Multiplication of two Q15 numbers.

Storing Q30 to 16-bit memory only requires the truncation as shown in Figure 4.5.

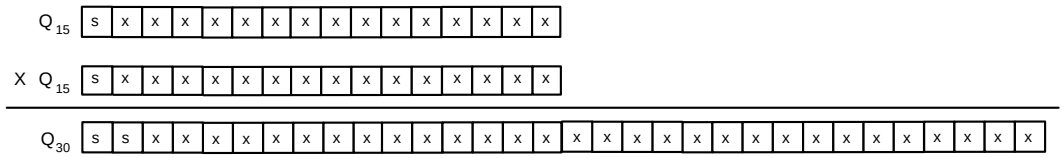

Figure 4.5: Q.30 truncation

Using Q15 format with numbers in the range -1 to 1, multiplication will always be in same range. In this quadrature mixer, multiplication is applied.

4.2 Low Pass Filter

Digital filters are specified by their impulse responses. The general constant-coefficient difference equation that relates the output to input is given by

$$y(n) = \sum_{k=1}^{N} a_k y(n-k) + \sum_{k=0}^{M-1} b_k (n-k) \tag{4.4}$$

so that there are N feedback paths and M feed-forward paths. Transforming Eq. 4.4 to the z-domain and rearranging, we can express the transfer function of

the filter as in (4.5). It is the general form for an infinite impulse response (IIR) filter.

$$H(z) = \frac{H(z)}{X(z)} = \frac{\prod_{k=0}^{M-1} b_0(1 - c_k z^{-k})}{\prod_{k=1}^{N}(1 - p_k z^{-k})} \tag{4.5}$$

For IIR filters, it is most common to use the analog filter prototypes and convert them with the bilinear transform. An alternative is to use other algorithms that are designed to minimize some error. These can be based on least squares optimization, etc.

4.2.1 IIR filter structure

In this project, the transposed form IIR filter mentioned above is implemented. Utilizing the same resources, data samples are applied in parallel to all the tap multipliers through pipeline registers. The input registers are not required, because high fan-out input signals can be handled by the FPGA architectures. The products are applied to a cascaded chain of registered adders, combining the effect of accumulators and registers. The order of tap coefficients must be reversed with the first tap closest to the output. This structure allows expansion of the number of taps required in a filter, since each "tap module" is identical. Since the structure is uniform, a single component can be designed and instantiated as many times as required by the number of taps.

In a fully parallel implementation of a filter, each tap has a dedicated multiplier. The tap data is an input of this multiplier, the other a constant coefficient. Since one input is a constant, these multipliers are called co-efficient components. Co-efficient components are efficiently implemented by storing pre-computed partial products of the fixed coefficient, thereby reducing the logic required as compared to traditional two-variable multipliers. As a result, better performance can be achieved. In hardware implementation, these partial products can be stored in

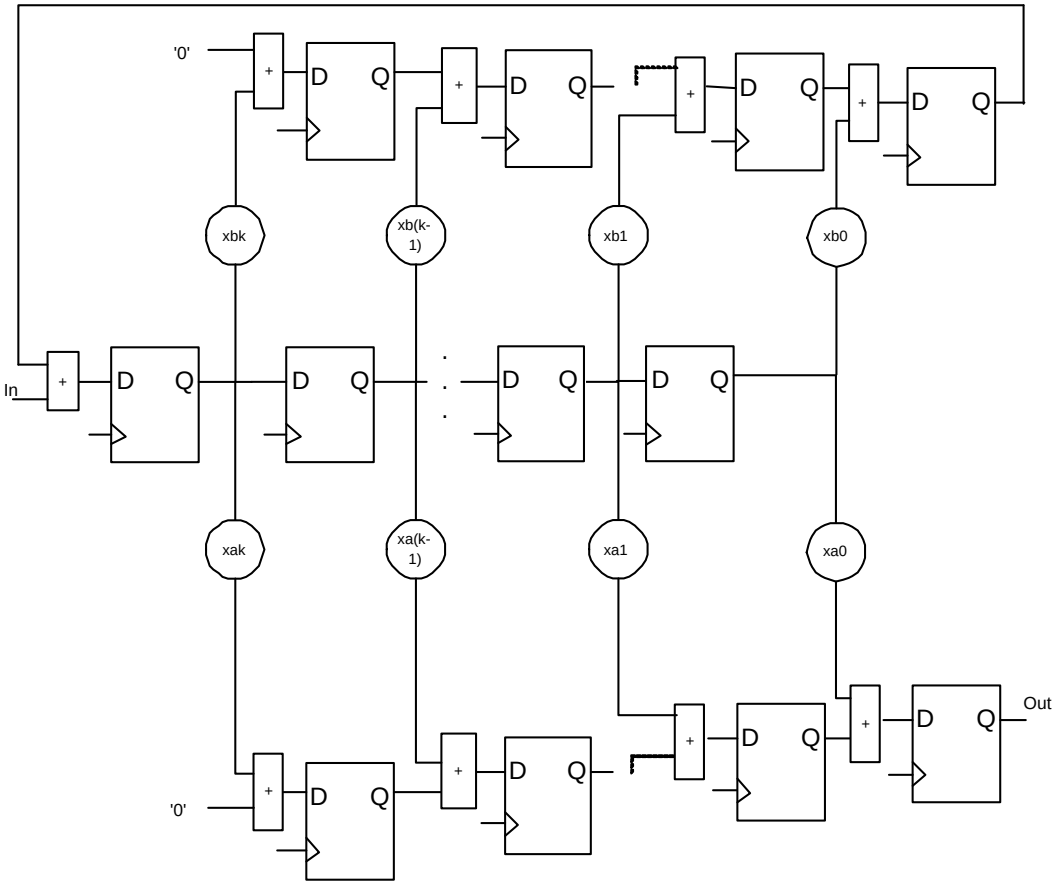

Figure 4.6: Transposed form IIR filters employing cascaded pipelined adders.

ROMs using the distributed memory. The 16-bit input sample is separated into four 4-bit nibbles. Each nibble acts as an input to the ROM in different cycles. These ROMs store the product of the constant coefficient k, and a factor with variable values that change from 0 through 15. The ROM contents are $0 \times k,\ 1 \times k,\ 2 \times k,\ 3 \times k,\ ,\ 15 \times k$. The word size in the ROM is:

(4-bit input nibble) $\times$ (14-bit coefficient) = 18 bits (ROM word size)

Essentially this ROM functions as a times table of the constant coefficient, k. The value read from this ROM based on its 4-bit input is added to another partial product stored in an adjacent ROM. As a result, co-efficient components are less than one-third the size of full multipliers. A co-efficient component block diagram is shown in Figure 4.7.

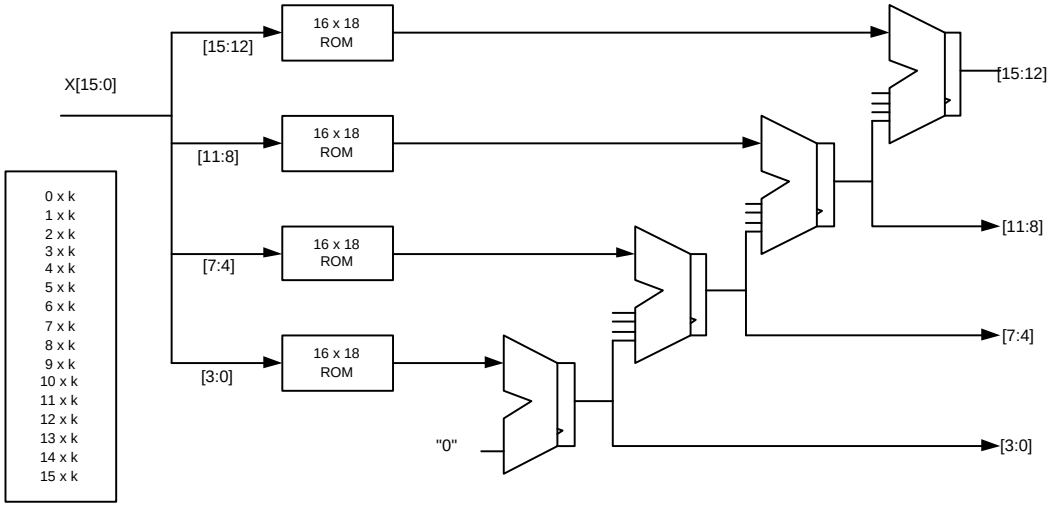

Figure 4.7: Coefficient component block diagram.

A co-efficient component would be ideal for unsigned inputs and coefficients. There are a couple of options for handling signed numbers. The first approach is to implement two ROM tables, one for the signed most significant bit (MSB) nibble and the other for the least significant bit (LSB) nibbles. This approach requires two separate ROM tables per tap. This is not an optimal solution. The second approach is to convert the signed sample input data into an unsigned magnitude word and a sign bit, using a 2's-complement module. When a negative word is detected, it is complemented, and the magnitude decodes a value from the same ROM table that a non-negative data would use. The multiplier output is a negative value, which is incorrect; however, the accompanying sign bit causes a subtract operation in the ADD/SUB module resulting in the correct sign and magnitude.

In order to handle signed inputs and coefficients, a 2's-complement component is used to convert negative numbers to positive. After all the operations, the final result is made positive or negative depending on the sign of the input and coefficient. The third approach uses three 2's-complement modules to handle both signed inputs and coefficients. This is used to avoid signed multiplication and addition. The operation of a co-efficient component multiplier implemented using a ROM is explained with the following example: 16-bit input: 0001 0010 0000

0100 (Decimal equivalent 4612) 14-bit coefficient: 00 0000 0000 0010 (Decimal equivalent 2)

The 16-bit input is separated into four 4-bit nibbles: "0001","0010", "0000", and "0100". All fifteen coefficient factors, $0 \times 2,\ 1 \times 2,\ 2 \times 2,\ ,\ 15 \times 2$ are stored with an 18-bit (14-bit $\times$ 4-bit) word size in the ROM. Each 4-bit nibble of the 16-bit input acts as an address to the ROM. The corresponding ROM content at this address is read.

First partial product = 00 0000 0000 0000 1000 (ROM contents at address "0100")

Second partial product = 00 0000 0000 0000 0000 (ROM contents at address "0000")

Third partial product = 00 0000 0000 0000 0100 (ROM contents at address "0010")

Fourth partial product = 00 0000 0000 0000 0010 (ROM contents at address "0001")

All the partial products are then added after shifting them appropriately (shown below):

```
                    00 0000 0000 0000 1000 First partial product
               00 0000 0000 0000 0000 0000 Second partial product
          00 0000 0000 0000 0100 0000 0000 Third partial product
+ 00 0000 0000 0000 0010 0000 0000 0000 Fourth partial product
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  00 0000 0000 0000 0010 0100 0000 1000(Decimal equivalent 9224)
```

Pipelining and resource sharing of adders can further enhance the performance of co-efficient component multipliers. An enhanced multicycle co-efficient component schematic is shown in Figure 4.8.

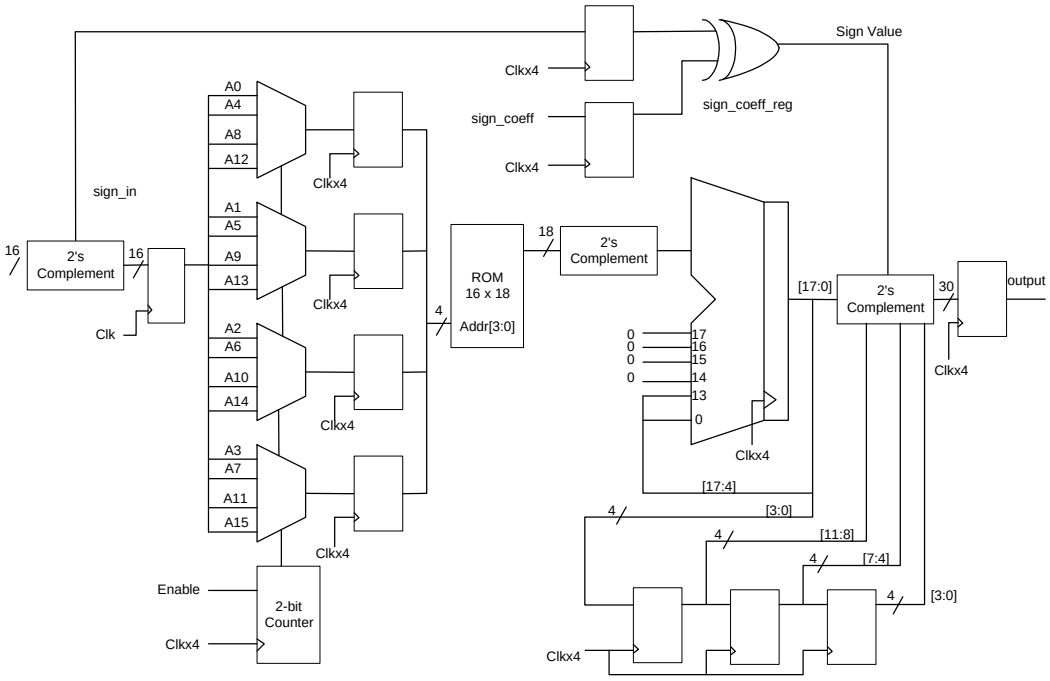

Figure 4.8: Multicycle co-efficient component implementation

The input sample arrives at clock frequency f_1 , while all the internal operations of the co-efficient component can be performed at a much higher frequency of $f_2(4f_1)$. Four muxes are used to select 4-bit input nibbles. A 2-bit counter clock operating at f_2 frequency acts as the select signal for these muxes. For every 4-bit input nibble, a corresponding value is read from the ROM and corresponding partial products are added after taking care of the required shift operations.

The complete filter is built by integrating the coefficient component multipliers, delay elements and adders as shown in Figure 4.9. Enlarged Figure 4.10, 4.11 and 4.12 give the details of the structure of the filter implemented. The VHDL source code can be found in Appendix C.

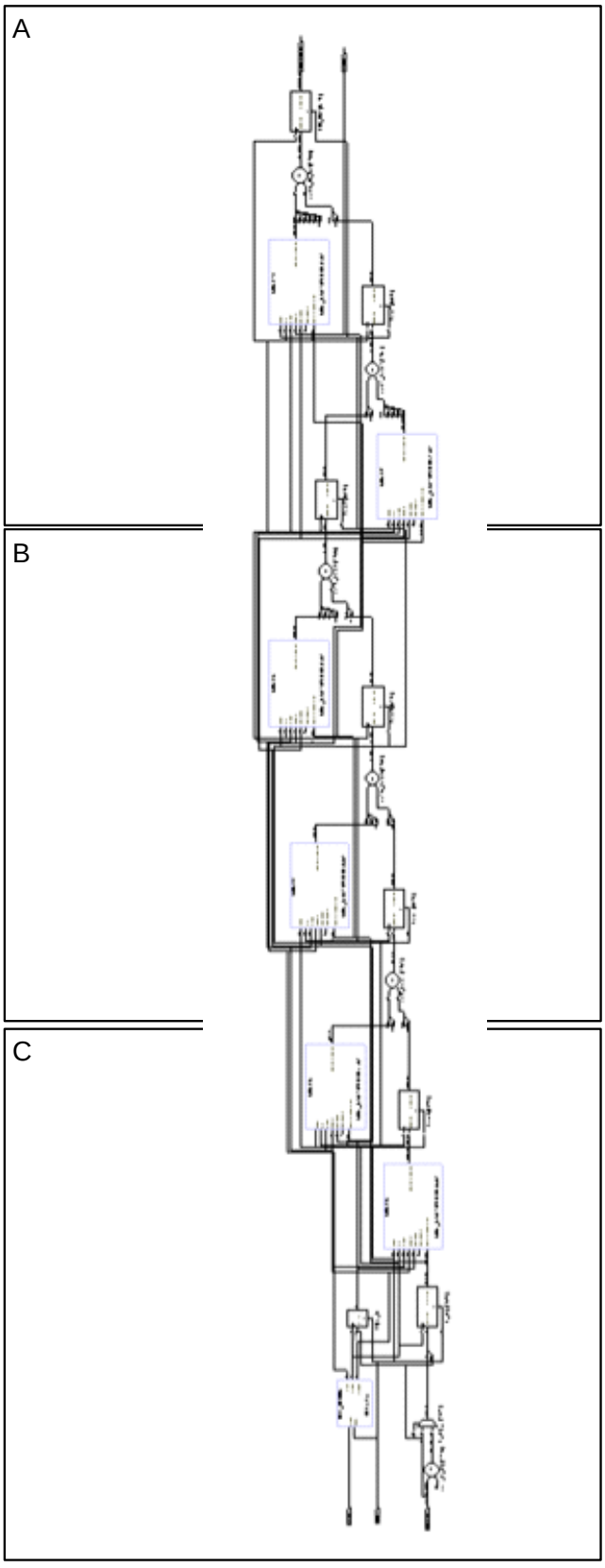

Figure 4.9: Screen capture of filter implementation from design software. (Clearer expanded diagrams in FIgures 4.10, 4.11 and 4.12)

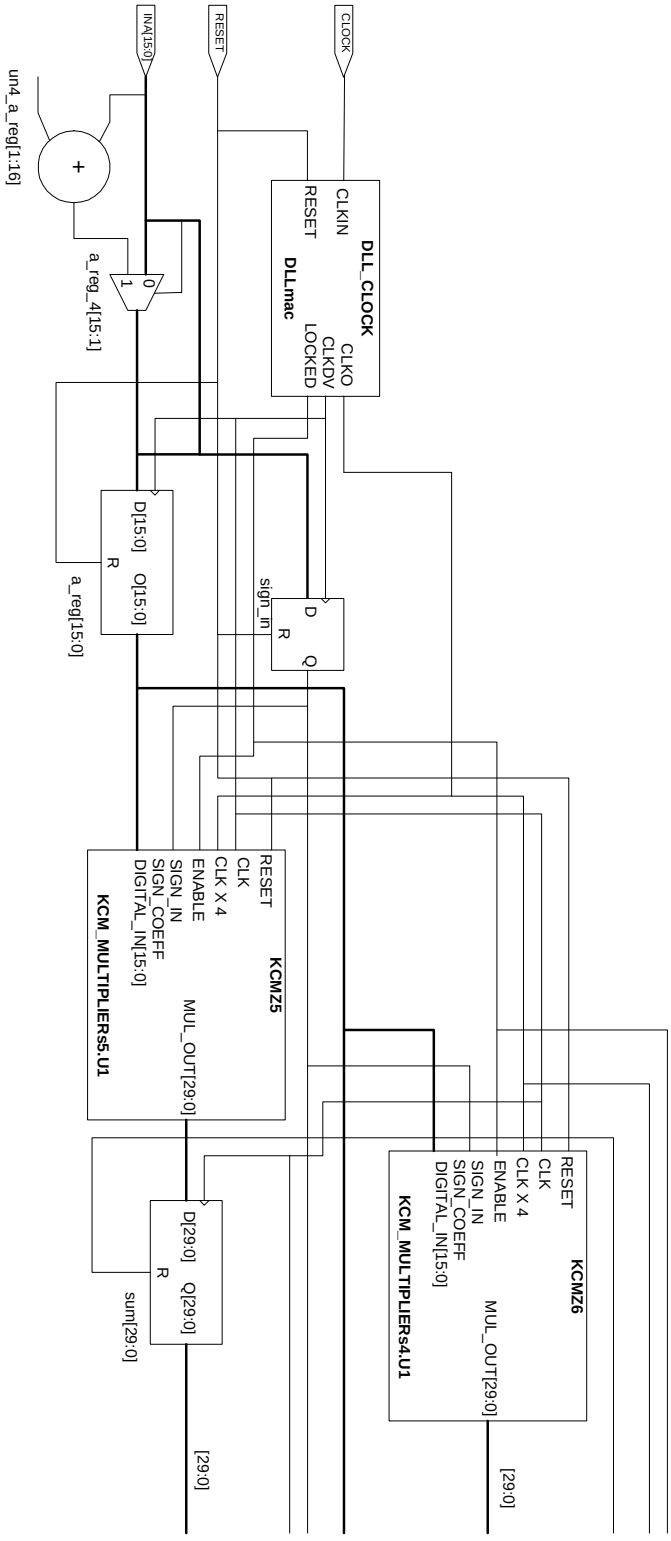

Figure 4.10: Part A of the filter implementation in Figure 4.9.

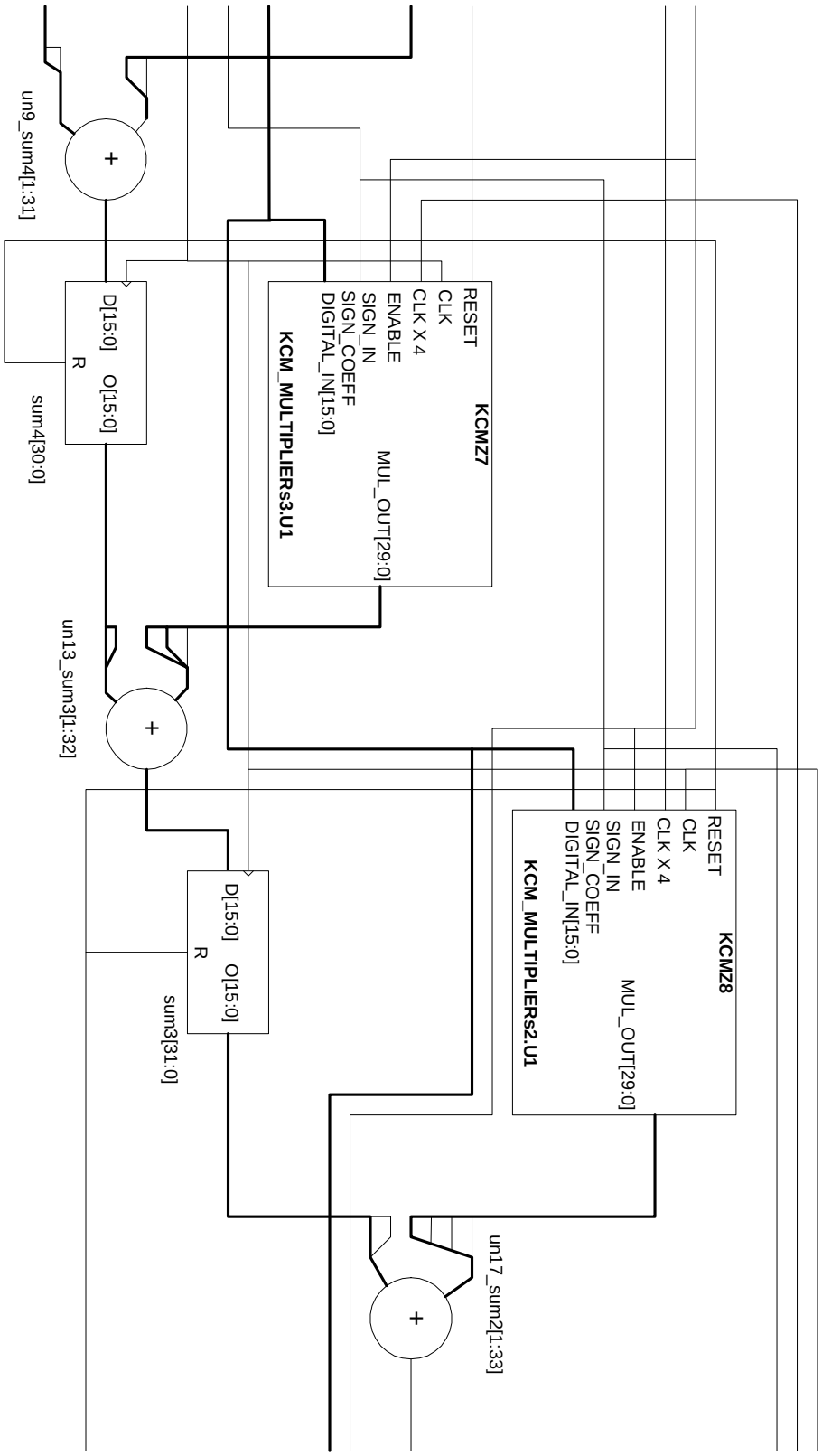

Figure 4.11: Part B of the filter implementation in Figure 4.9.

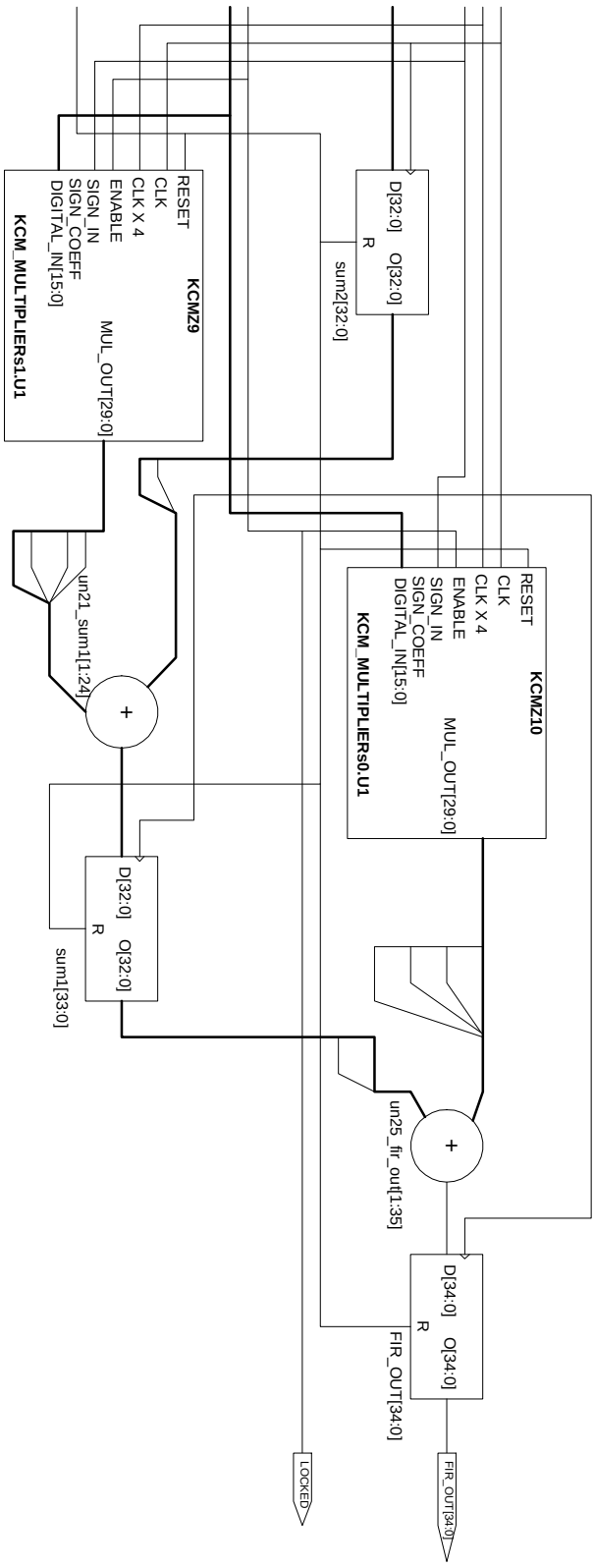

Figure 4.12: Part C of the filter implementation in Figure 4.9.

4.2.2 Delayed locked loop (DLL)

As discussed above, multicycle co-efficient component uses two clocks of frequency f_1 and f_2, where $f_1 = f_2/4$ or $f_2 = 4 \times f_1$. A clock phase deskew and clock manipulation circuitry is needed. There are two basic types of circuits remove clock delay: PLLs, and DLLs. In addition to the primary function of removing clock distribution delay, DLLs and PLLs typically provide some additional functionality such as frequency synthesis (clock multiplication and clock division) and clock conditioning (duty cycle correction and phase shifting). Multiple clock outputs can also be de-skewed with respect to one another, to take advantage of multiple clock domains.

A DLL consists of 3 main blocks: the delay line, the phase detector, and the delay control logic as shown in Figure 4.13. The delay line is composed of a series of delay elements implemented using an invertor and a multiplexor. The delay line is used to delay the input clock *clkin* using a number of delay elements as specified by the shift register in the delay control logic. The delayed output clock *clkout* is being feedback to the phase detector for comparison with the original input clock *clkin*. The phase detector will indicate to the delay control logic whether to increase or to decrease the clock delay via the signals $shiftleft$ and $shiftright$ respectively. The shift clock of the shift register is divided (by 4) from the main clock (or *clkin*) so as to have a stable shift operation of the delay line. In other words, the adjustment of the delay line only happens once in every 4 clock cycles. When *clkout* is in-phase with *clkin*, both $shiftleft$ and $shiftright$ signals will be de-asserted and this will indicate that the DLL is locked. The signal lock will indicate the lock status of the DLL, whereas the bus signals *lockvalue* will indicate the number of delay elements used to delay *clkin* by one clock period.

Using the DLL design described above as the Master DLL, a master-slave delay line architecture is separately implemented as shown in Figure 4.15. The purpose of the Master DLL is to compute the number of delay elements or *lockvalue* required to delay any signals by one clock period duration. With this value made known

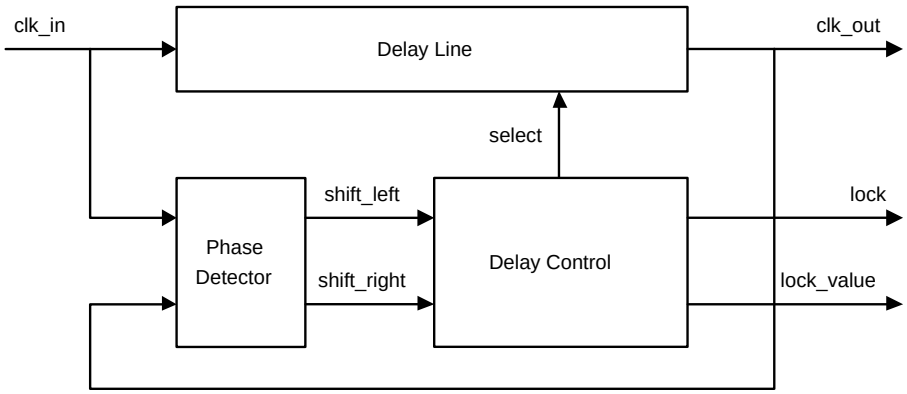

Figure 4.13: Delayed lock loop architecture.

to the slave delay line control logic, it is possible to delay any of the slave clock signals *slaveclkin* by any required percentage of a clock period. The user can program the delay percentage in terms of $N/128$ of the clock period. The slave delay line control logic will compute the number of delay elements required to delay *slaveclkin* using the formula $lockvalue \times N/128$ and the slave delay line will be adjusted accordingly. For instance, if the Master DLL is locked with a *lockvalue* of 200, and the slave delay line is programmed to delay the slave clock by 1/4 (ie. N = 32), then 50 delay elements will be used for the slave delay line. The advantage of this architecture is highly flexible programmable, from delay stage to lock status and its periodic calibration function.

Since the Xilinx FPGA devices (Virtex series) provide up to eight fully digital dedicated on-chip Delay-Locked Loop (DLL) circuits, which provide zero propagation delay and low clock skew between output clock signals distributed throughout the device, the above DLL implementation is not directly adapted. This module can be used for future ASIC integration as a reference.

In the Xilinx FPGA devices, each DLL can drive up to two global clock routing networks within the device. The global clock distribution network minimizes clock skews due to loading differences. By monitoring a sample of the DLL output clock, the DLL can compensate for the delay on the routing network, effectively eliminating the delay from the external input port to the individual clock loads

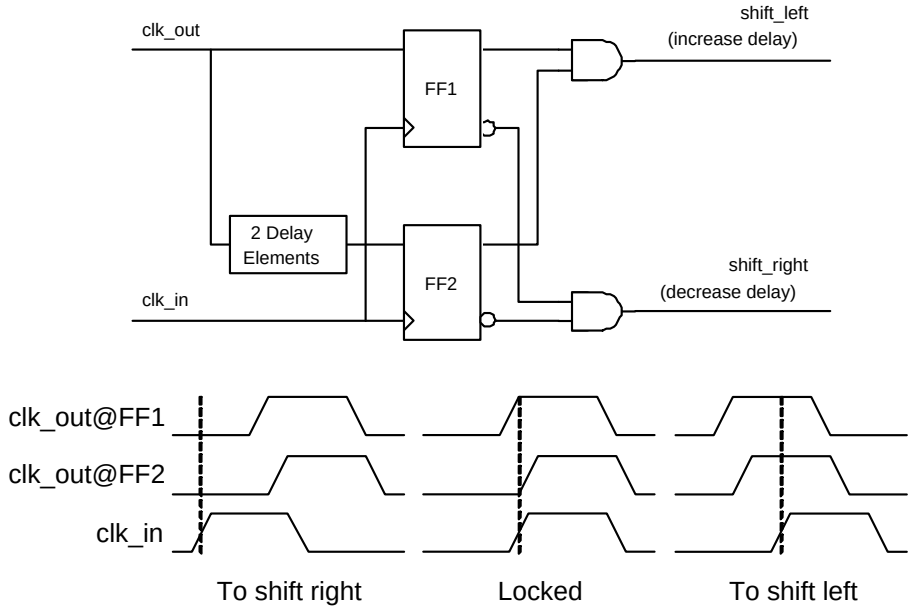

Figure 4.14: Delayed lock loop phase detector.

within the device.

In addition to providing zero delay with respect to a user source clock, the DLL can provide multiple phases of the source clock. The DLL can also act as a clock doubler. Two DLLs can be connected in series to increase the effective clock multiplication factor to four. So the on-chip DLL will be utilized directly instead of building it up from scratch. A logic symbol of Xilinx DLL is shown in Figure 4.16.

4.3 Mixed Demodulator

Figure 4.17 shows the architecture of the mixed demodulator. To implement the mixed demodulator, one approach is to follow the sequence as shown in Figure 4.17. It means tangent value should be gotten first, then to derive the angle via arctangent function with avoiding scaling and overflow problems (divided by Tk_{FM} factor). For arctangent function implementation, a straightforward method is to

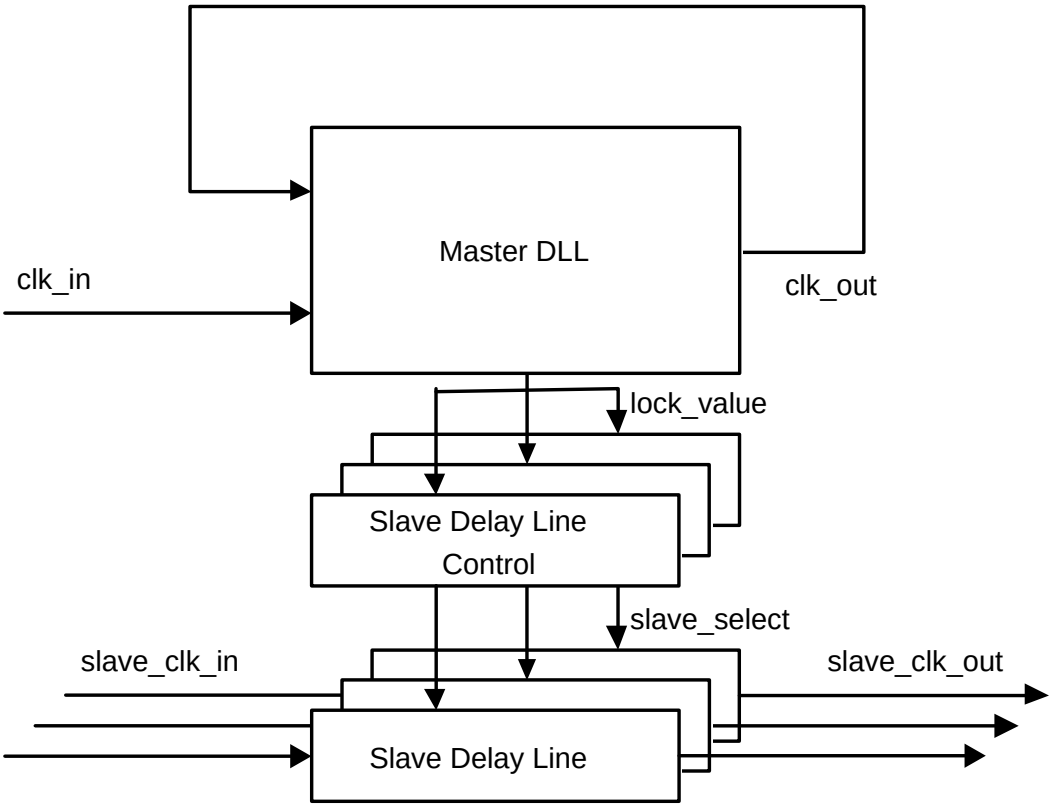

Figure 4.15: Master-slave delay line architecture.

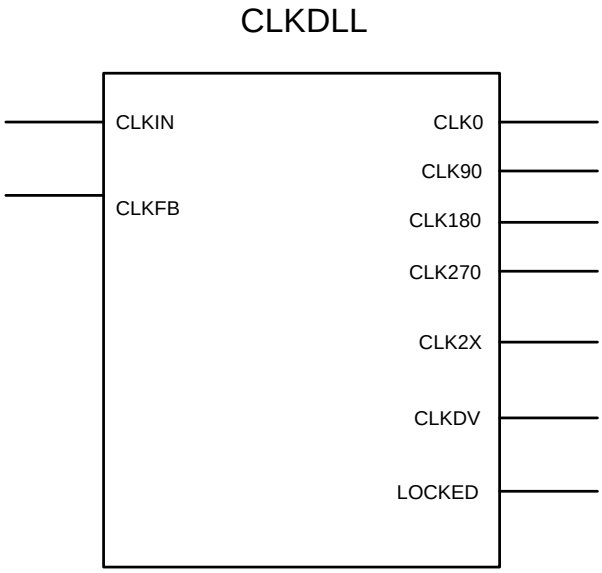

Figure 4.16: Standard DLL symbol CLKDLL [53]

use Lookup Table. The factor $1/Tk_{FM}$ can be included also. Therefore the lookup table fulfils the equations:

$$out = \frac{1}{Tk_{FM}} arctan(in)$$

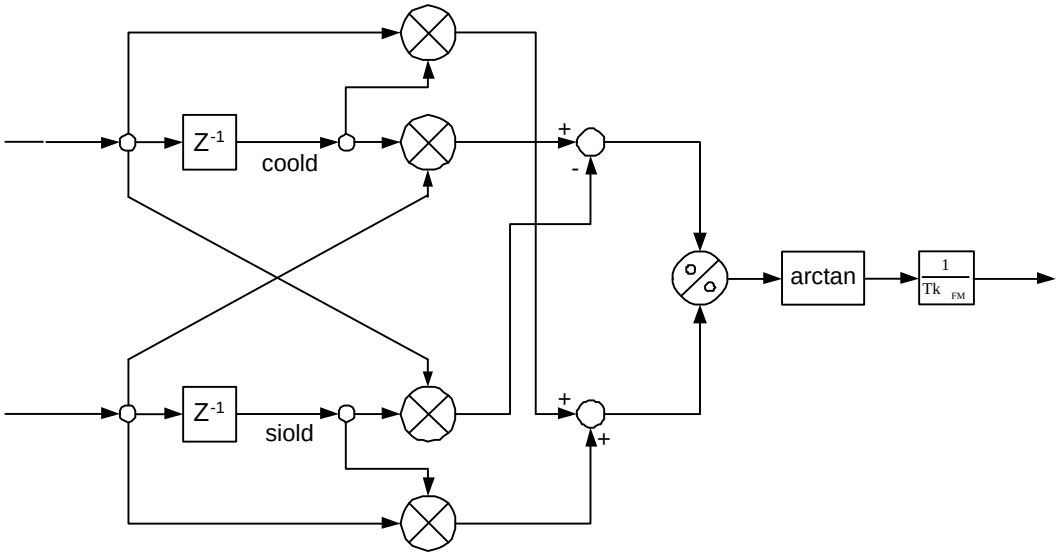

Figure 4.17: Block diagram of mixed demodulator

But to get the tangent value, a division calculation need to be involved. To implement a divider, a 16-bit to 16-bit divider can be directly mapped. But it will consume more gates and more clock cycles to get the required resolution. Another method used widely in communication system is to implement a logarithmic arithmetic unit. It can convert division into subtraction operation. The speed can be improved significantly. But for this project, this method introduces the conversion between number and logarithmic values. Another two tables must be added. This will affect the resolution and area.

Another approach to implement the mixed demodulator is that the division and arctangent calculation is considered together. If considering the two factors of division as two input vectors to calculate arctangent, it is exactly what the CORDIC Vector Rotation does.

4.3.1 The Details of CORDIC Implementation in an FPGA

In this project, the VectorCordic and the RotationCordic performs the vectoring and rotation mode of the CORDIC operations to convert the Cartesian vectors to Polar forma and vice versa.

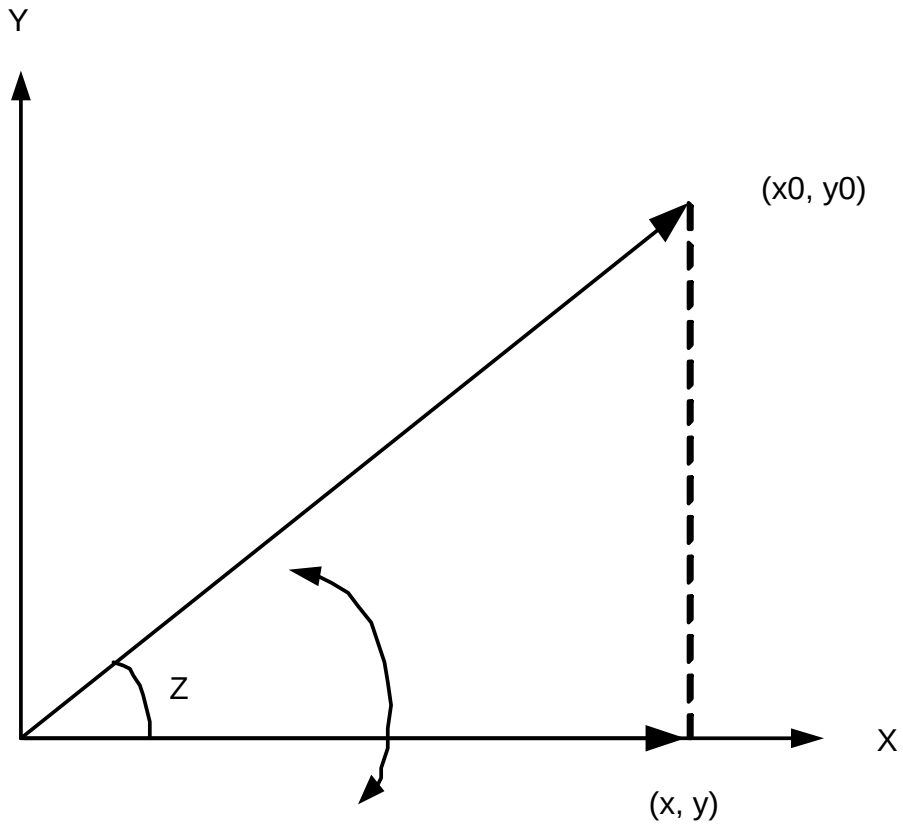

Figure 4.18: The Vectoring Mode

Figure 4.18 describes the vectoring mode of the CORDIC operations. Given a Cartesian vector (x_0, y_0), the vector is projected onto the x-axis after eight iterations. In each iteration i, the vector is rotated at an angle z_i in clockwise or anti-clockwise depending on the direction of the vectors vertical displacement in the previous iteration. The computation of the vector and the angle of rotation can be described with the following:

$$X_{i+1} = X_i + S_{yi}Y_{ishifted} \tag{4.6}$$

$$Y_{i+1} = Y_i - S_{yi}X_{ishifted} \tag{4.7}$$

$$Z_{i+1} = Z_i + Z_{iconstant} \tag{4.8}$$

The vector (X_i, Y_i) and Z_i represents the vector and angle of rotation in the ith iteration, S_{yi} is the polarity due to the direction of the y_is vertical displacement, $x_i$shifted/$y_i$shifted are 1-bit shifted values of X_i and Y_i, Z_i signifies the cumulative angle and Z_i constant represents a constant value to be added to the cumulative angle in the ith iteration. Thus after the eighth iteration, the y value will be equal to 0, the X and Z will represent the pseudo magnitude and angle of the original vector in polar form. The true value of the magnitude requires the incorporation of a scaling factor, which has been included in the back-mapping coefficients.

Figure 4.19 illustrates the architecture of VectorCordic. The architecture is controlled by a 8-state controller (ControllerState) and consists of four main units. The VectorCordic Iteration performs the main vectoring mode CORDIC computations to obtain the X, Y and Z values. It obtains the input parameters of the previous iterations from Register and the shifted parameters of the previous iteration from the shift register in Cordic Shifter. To compute the angle of rotation Z in each iteration, a different constant is required. These values can be obtained from VectorCordic Z Constant Register and Figure 4.20 describes the constant $Ziconstant$, which are hard-coded in the unit.

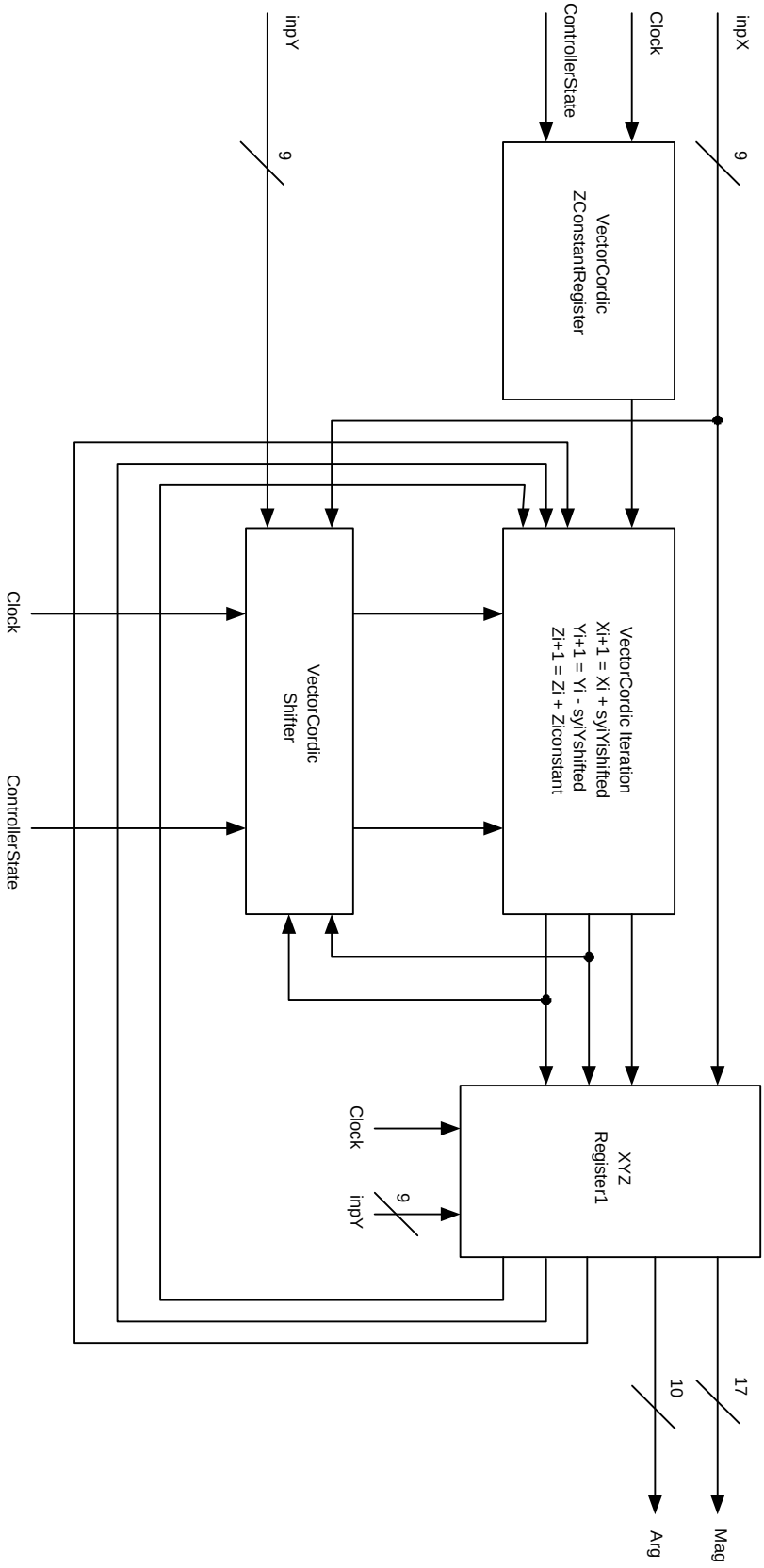

Figure 4.19: The Architecture of the Vectoring Mode

Iteration i	Ziconstant
0	11101101011000
1	01111101011011
2	00111111101010
3	00011111111101
4	00001111111111
5	00000111111111
6	00000011111111
7	00000001111111

Figure 4.20: The Constant Ziconstant for the Vectoring Mode

Figure 4.21 describes the rotation mode CORDIC to convert the polar representation back to the Cartesian form. Given an initial magnitude x_0 and angle z_0, the vector is rotated for sixteen iterations until the final Cartesian vector is obtained. In each iteration i, the vector is rotated at an angle Z_i in clockwise or anti-clockwise depending on the direction of the angles rotation in the previous iteration. The computation of the rotation mode can be described with the following:

$$X_{i+1} = X_i - S_{zi} Y_{ishifted} \tag{4.9}$$

$$Y_{i+1} = Y_i + S_{zi} X_{ishifted} \tag{4.10}$$

$$Z_{i+1} = Z_i - Z_{iconstant} \tag{4.11}$$

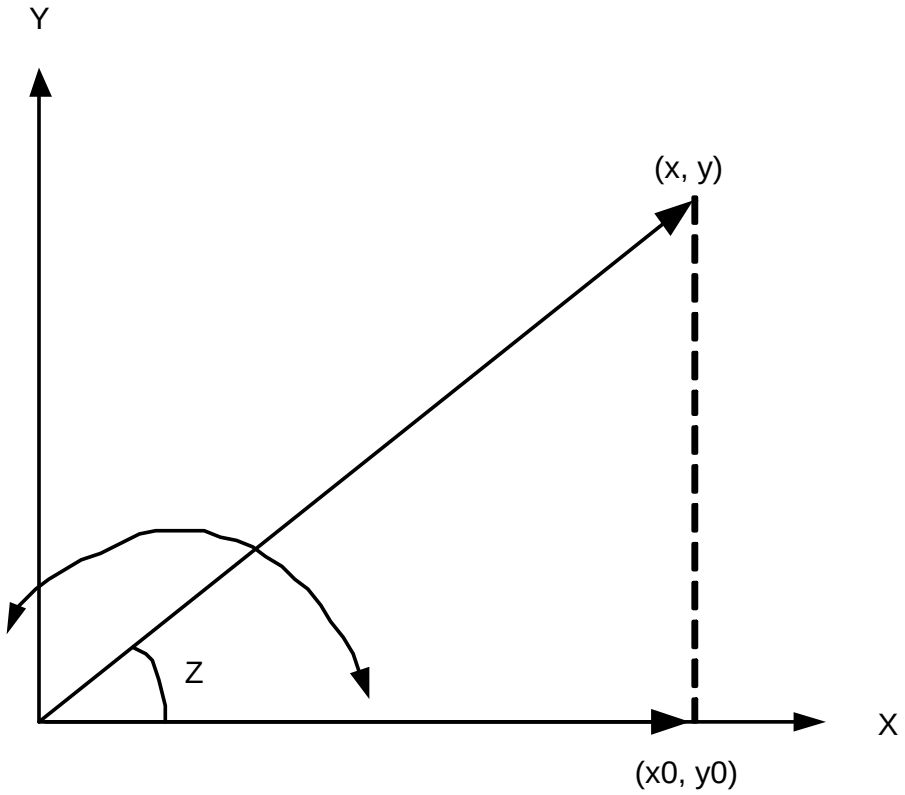

Figure 4.21: The Rotation Mode

The vector $(x_i,\ y_i)$ and z_i represents the vector and angle of rotation in the ith iteration, S_{zi} is polarity due to the rotational direction of z_i, Xishifted/Yishifted are 1-bit shifted values of X_i and Y_i, Z_i signifies the decremented angle and $Ziconstant$ represents a constant value to be added to the cumulative angle in the ith iteration. Thus after the sixteenth iteration, x and y will represent the Cartesian vector of the input polar representation.

Figure 4.22 describes the architecture of RotationCordic. The architecture is controlled with a 8-state controller. Since, the number of iterations required to perform the rotation mode is 16 and each pipeline stage in the Correction Unit is restricted to 8 clock cycles, the operations must be performed in two pipeline stages. The RotationCordic performs the main vectoring mode CORDIC computations to obtain the x, y and z values. There exist two such units (stage 1 and stage 2) to perform the computations in a two-stage pipeline fashion. For example, after the first stage

has performed 8 iterations of the rotational mode, the second stage will take over to complete the last 8 iterations. Meanwhile, the first stage can process a new set of conversion. The RotationCordic Iteration stages obtain the input parameters of the previous iterations from Register and the shifted parameters of the previous iteration from the shift registers in RotationCordic Shifter. To compute the angle of rotation Z in each iteration, a different constant is required. These values can be obtained from RotationCordic Z Constant Register.

In order to verify the implementation, both vector CORDIC and rotation CORDIC have been modelled using VHDL (VHSIC Hardware Description Language) and synthesized with Synplcity Synplify pro 7.2 and mapped with Xilinx ISE 4 targeted at the Xilinx Virtex II XC2V500 FG264 FPGA device. The area-time analysis of the experimental application is obtained from the .srr file as shown in Table 4.1, which summarized the resource utilized and process time. The VHDL source codes are listed in Appendix C.

Table 4.1: Summary of Resources Consumed

Parameters	Vector CORDIC	Rotation CORDIC
Area (SLICEs)	328	359
Time	181.75 ns	196.54 ns

As discussed earlier, CORDIC algorithm is iterative in nature. It will require more clock cycles to achieve the resolution. Since DLL concept (4 times clock frequency) is introduced in the project, this approach is feasible.

4.4 Hardware Implementation

The mixed demodulator is implemented in Xilinx FPGA Virtex 2 XC2V500. The source codes are attached in appendix C.

4.4.1 System Architecture

Figure 4.23 and Figure 4.24 outline the implementation stage of the demodulator. A master clock is provided to the whole system. For the quadrature mixer, two tables are implemented in Q15 format for sine/cosine values respectively. Transposed low pass IIR filter is adopted together with constant coefficient multiplier components inside. CORDIC algorithms are used for computing arctangent values. The clock of IIR filters and CORDIC cores is 4 times of system clock which is derived from DLL output. The data format used in this implementation is 16 bit.

In order to verify the performance, the mixed demodulator architecture has been modelled using VHDL and synthesized with Synplcity Synplify pro 7.2 and mapped with Xilinx ISE 4 targeted at the Xilinx Virtex II XC2V500 FG264 FPGA device. The area-time analysis of the experimental application is obtained from the .srr file as shown in Table 4.2, which summarized the resource utilized and process time. The VHDL source codes are listed in Appendix C.

Table 4.2: Resource utilization of the implementation

Resources	Values
Area (number of slices)	2,427 out of 3,072 (79%)
Timing for whole process (from input of quadrature mixer to output of IIR filter)	15,621 ns

It can be observed that the mixed demodulator architecture requires only 15.6μs to finish the data process from input of quadrature mixer to final output of IIR filter.

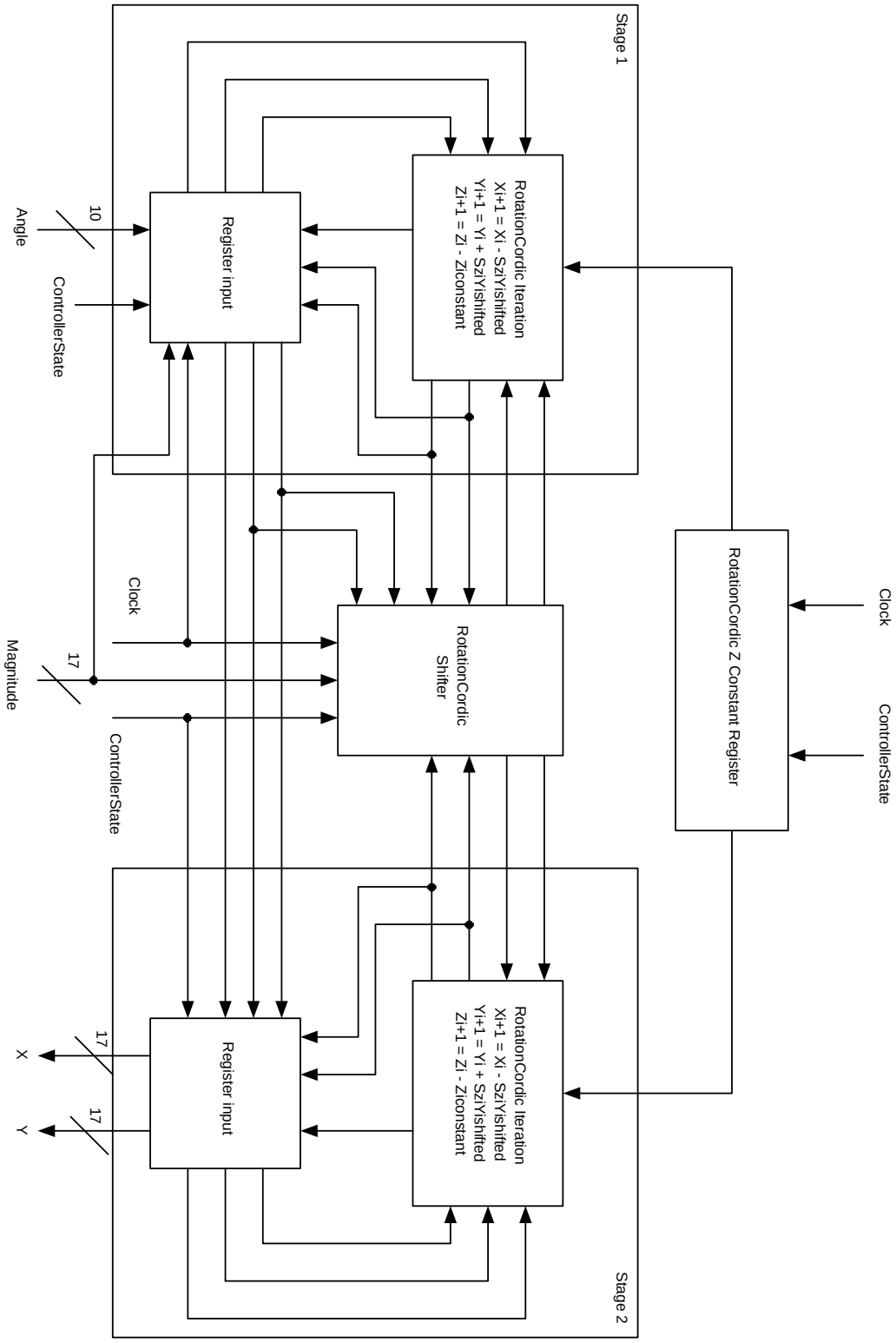

Figure 4.22: The Architecture of Rotation Mode

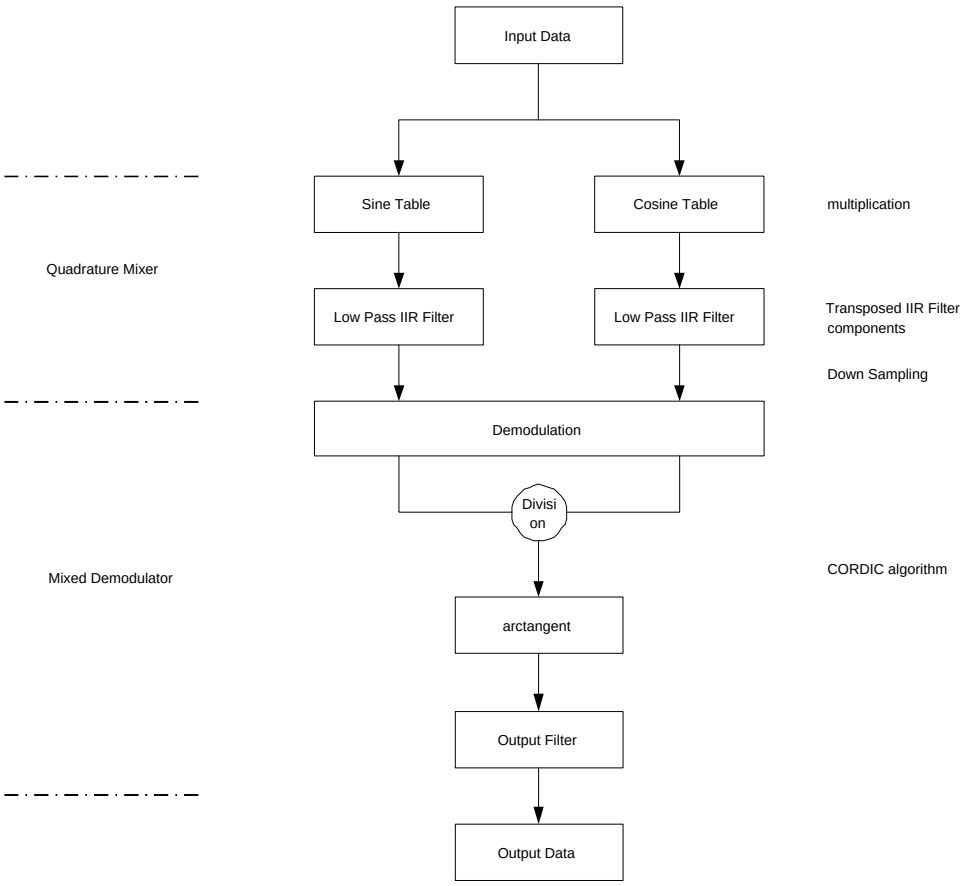

Figure 4.23: System architecture of the mixed demodulator

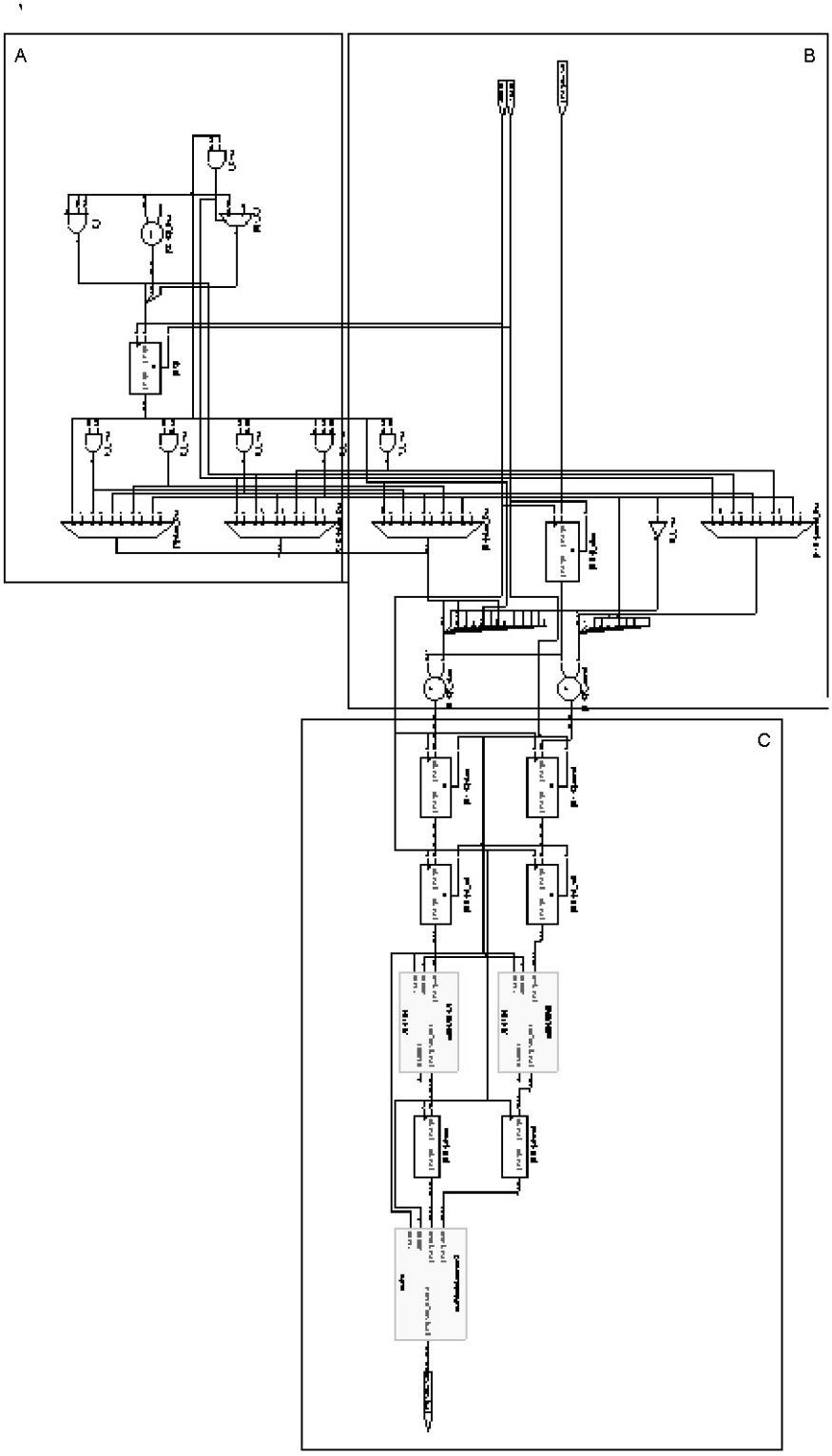

Figure 4.24: Screen capture of the mixed demodulator implementation. (Clearer expanded diagrams can be found in Figures 4.25, 4.26 and 4.27.)

The enlarged Figures 4.25, 4.26 and 4.27 give the details of the structure of the mixed demodulator implemented.

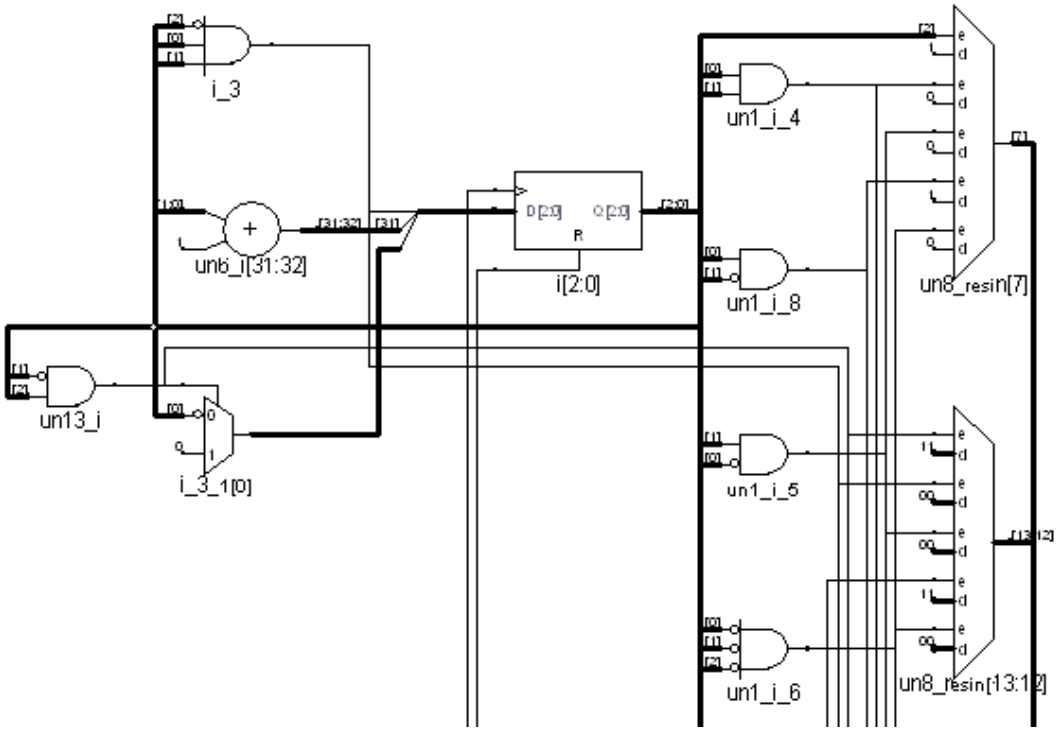

Figure 4.25: Part A of the mixed demodulator implementation in Figure 4.24.

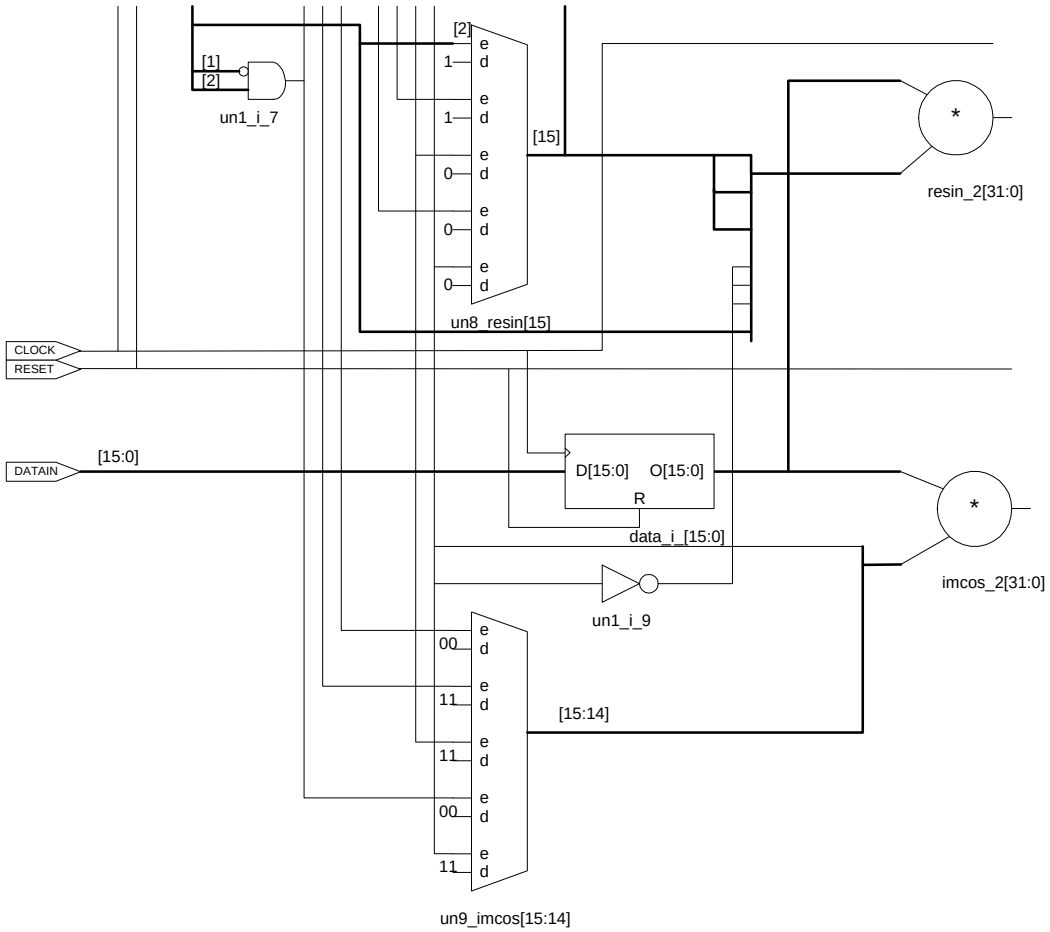

Figure 4.26: Part B of the mixed demodulator implementation in Figure 4.24.

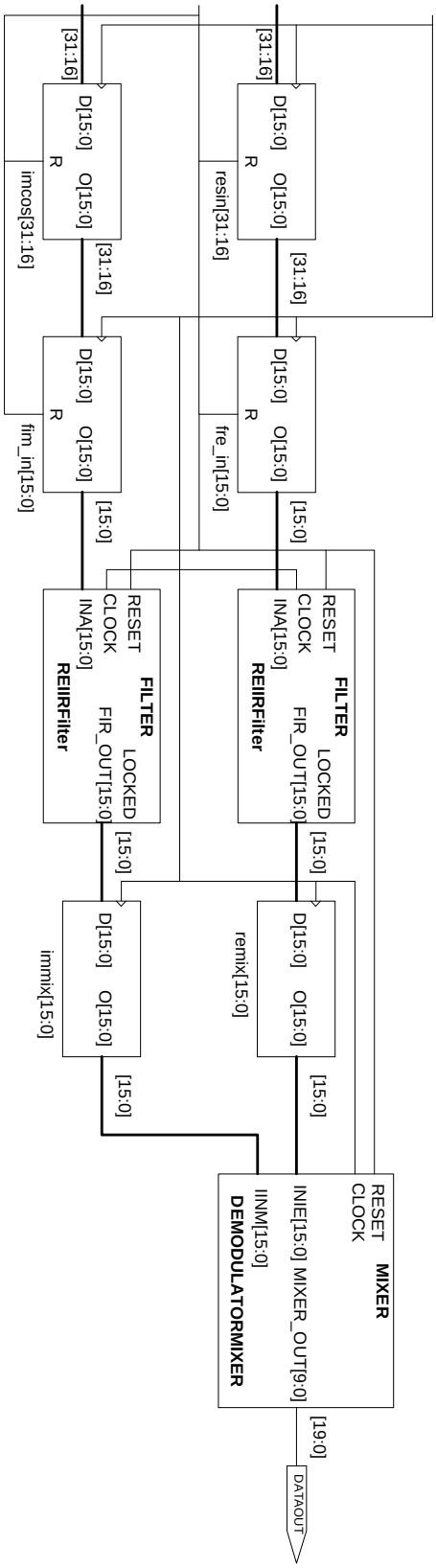

Figure 4.27: Part C of the mixed demodulator implementation in Figure 4.24.

4.5 Summary

The structure of the full digital FM demodulator and its hardware implementation are presented in this chapter. In particular, the implementation details of three main parts: quadrature mixer, low pass filter and mixed demodulator and whole demodulator are shown. The design is implemented on a Xilinx FPGA and its performance evaluated. Results of the evaluation are presented in the next chapter.

Chapter 5

Measurements and Performance Analysis

The performance of the design described in the previous chapter is evaluated through physical measurements. The results are compared with the DSP implementation reported in Schnyder and Hall's report [39]. In order to make a fair comparison, we adopted same specifications for the test signals as [39].

5.1 Specifications of Input Signals

The signals used for testing have the following specifications.

1. The modulated signal S_{FM} is frequency limited at an intermediate frequency of 10.7 MHz. The antenna, tuner, and bandpass filter are given and do not covered by this work. The message signal S_N is a speech signal from 300 Hz to 3400 Hz. The amplitude is normalized to one. The input FM signal is generated with the function generator in this project, with the following parameters:

   - Carrier frequency $f_T = 10.7$ MHz.
   - Frequency deviation $\Delta F = 3$ kHz.

- Message signal frequency range is $100 - 3700$ Hz.

2. The FM-signal S_{FM} has a bandwidth of 12.5 kHz and carrier frequency of 10.7 MHz. The amplitude is also normalized to one.

5.2 The Setup of Measurement

To do the measurement, the test equipment is set up as shown in Figure 5.1. The demodulator is implemented on a Xilinx FPGA prototype board. The full list of equipment used in the measurements can be found in Appendix A.

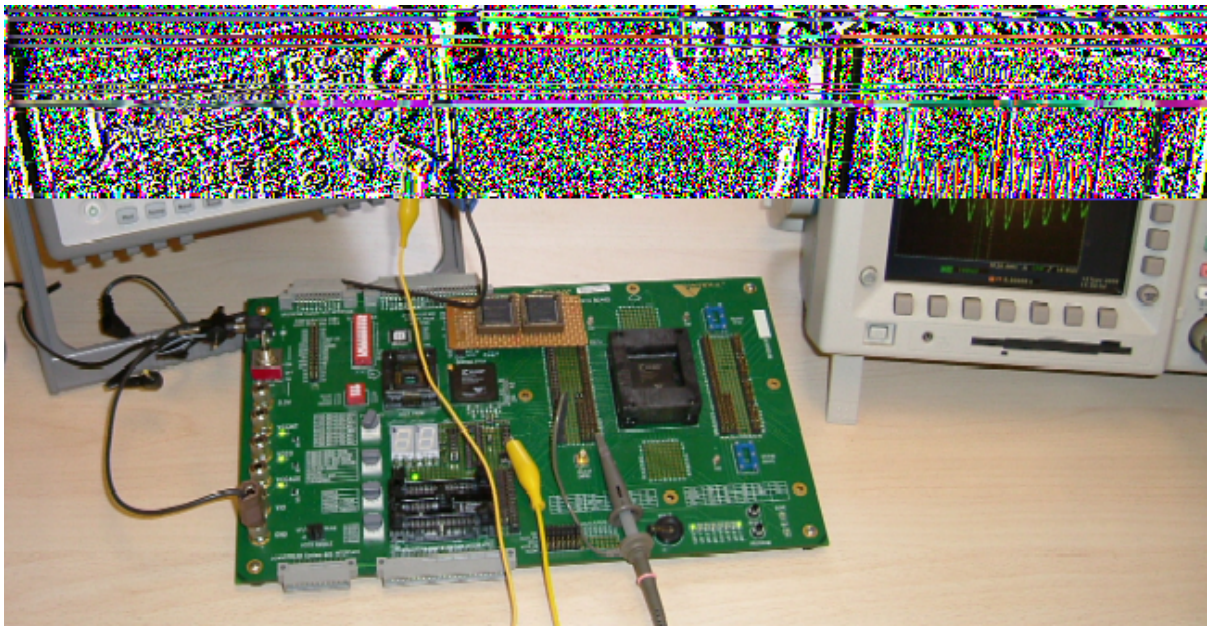

Figure 5.1: The equipment setup.

5.3 Analysis of Measurements

5.3.1 Spectrum Analysis

The spectrum of a signal gives a general statement about the signal. The Fast Fourier Transformation (FFT) of the measured signal is calculated.

Compared to the spectrum of a pure sine signal, the noise floor of the demodulated signal with mixed demodulator is mainly introduced by the demodulation algorithm and not by the used devices. even compared with the spectrum of the same signal over the maximal range the equipment used can handle, the Noise reduction between 3200 Hz and 4000 Hz is introduced by the IIR interpolation filter. The noise above 4000 Hz is the noise level introduced by the devices. This noise level is more than 20dB lower than the noise level introduced by the demodulation algorithms and will therefore not distort the subsequent measurements.

5.3.2 Quality of Signal

The distortion factor k shows the rate of nonlinear distortions in a signal, which is normally used to indicate the signal quality. It is defined as:

$$k = \sqrt{\frac{P_{total} - P_f}{P_{total}}} \tag{5.1}$$

The power is calculated over a time period. P_{total} is the power of the whole signal. To calculate the power $P_{total} - P_f$, the signal is first filtered with a band-stop and calculated subsequently. The calculated factor corresponds to the harmonic distortion if there are just harmonics and no noise. Otherwise, both harmonics and noise are included. Therefore, the calculated factor is called SINAD (Signal, Noise and Distortion).

The equipment NCI A2-D has a $THD + N$ Function, which measures the total Harmonic Distortion and noise by inserting a band-reject (notch) filter into the signal path (Figure 5.2). This figure is referred from Schnyder and Hall's report [39]

The $THD + N$ value is calculated according to

$$THD + N = \frac{(Distortion + Noise)}{Signal + (Distortion + Noise)}$$

This definition is equal to the concept of SINAD used in Schnyder and Hall's

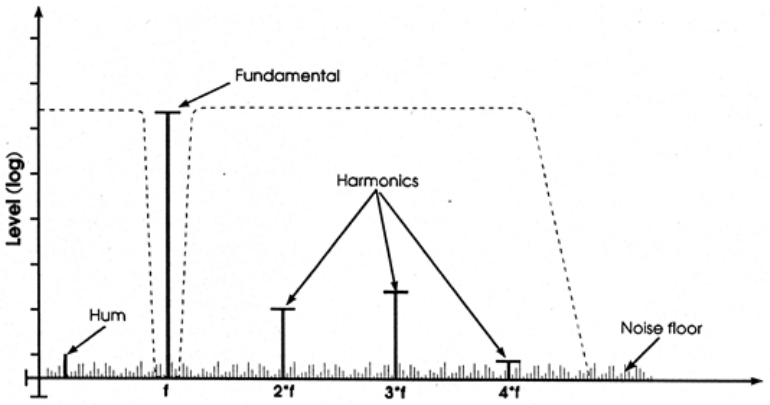

Figure 5.2: Principle of THD + N measurement.

report [39], which was measured to gauge the signal quality. Therefore THD+N values are referred to as SINAD in the remainder of this chapter to keep the same reference to the DSP implementation.

The quality of the output signal depends a lot on the accuracy of the implementation of the arctangent function. Figure 5.3 shows the plot of the SINAD of FPGA implementation. The measured data are given in Table B.1 in Appendix B.

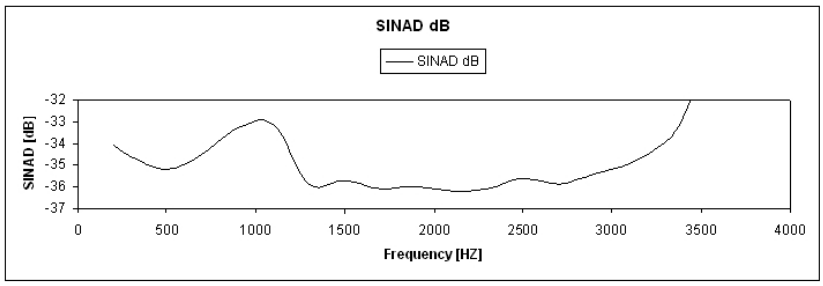

Figure 5.3: SINAD of mixed demodulator implemented in FPGA.

Comparing with the same implementation in DSP processor, the implementation in FPGA has shown nearly the same behavior as shown in Figure 5.4. The result implemented in DSP processor is referred from Schnyder and Hall's report [39]. Both implementations show a slight decrease of signal quality around the frequency of 1000 Hz.

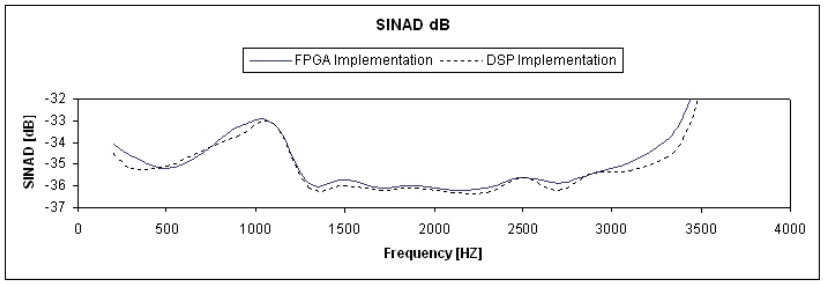

Figure 5.4: SINAD comparison of mixed demodulator implemented in FPGA and DSP.

5.3.3 Robustness Analysis

To test the robustness of the implementations noise is added to the FM signal before it is applied to the demodulator and the signal-to-noise ratio (SNR) is measured at the demodulated signal. The noise is generated with signal generator (Agilent 33250A).

The SNR can not be measured directly. But it can be determined from the $THD+N$ measurement using NC10 (see above), when everything other than the signal (S) is considered as noise (N_T), including the distortions (D).

$$\frac{1-(THD+N)}{(THD+N)} = \frac{1-\frac{D+N}{S+D+N}}{\frac{D+N}{S+D+N}} = \frac{S+D+N-(D+N)}{D+N} = \frac{S}{D+N} = \frac{S}{N_T} \tag{5.2}$$

The implementations are tested with two different noise levels also to align with DSP implementation done by Schnyder and Hall for comparison. For the first the S/N of the FM signal is set to 20dB and for the second it is set to 15dB. For the mixed demodulator algorithm the robustness of the implementation is measured. The result is shown in Figure 5.5, when the input FM signal has a S/N of 20dB. All the measured data are shown in Table B.2 in Appendix B.

The result is shown in Figure 5.6, when the input FM signal has a SNR of 15dB.

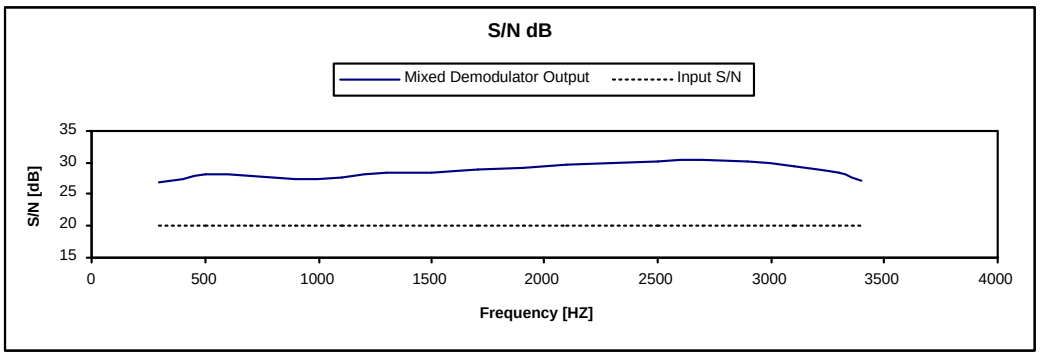

Figure 5.5: S/N of the mixed demodulated signal with a S/N of 20dB at the input FM signal

The measured data are shown in Table B.3 in Appendix B. Both figures also show the SNR of the input FM signal as a reference.

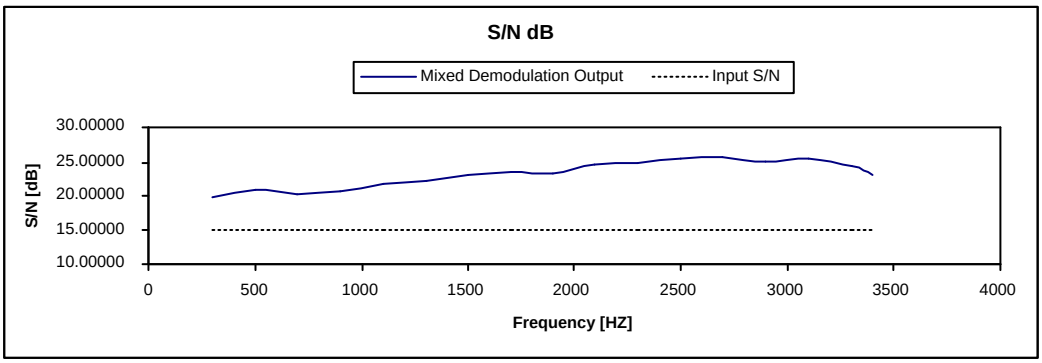

Figure 5.6: SNR of the mixed demodulated signal with a SNR of 15dB at the input FM signal.

It can be seen that all the implementations have an almost identical behavior. The SNR is improved compared to the input ratio for all frequencies. It is slightly better for higher frequencies than for lower ones.

If comparing with the implementation in DSP processor, the robustness implemented in FPGA shows a slightly worse. Both SNR comparison results of 20dB and 15dB are shown in Figure 5.7 and Figure 5.8. The result implemented in DSP processor is referred from Schnyder and Hall's report [39].

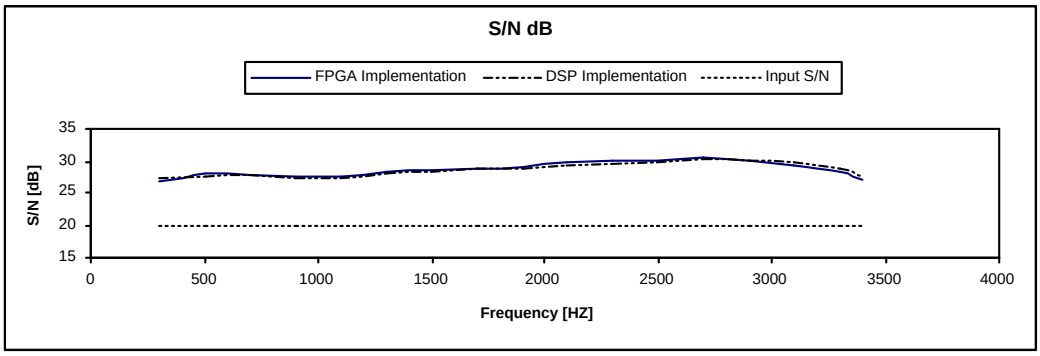

Figure 5.7: Comparison of the mixed demodulated signal with a SNR of 15dB at the input FM signal.

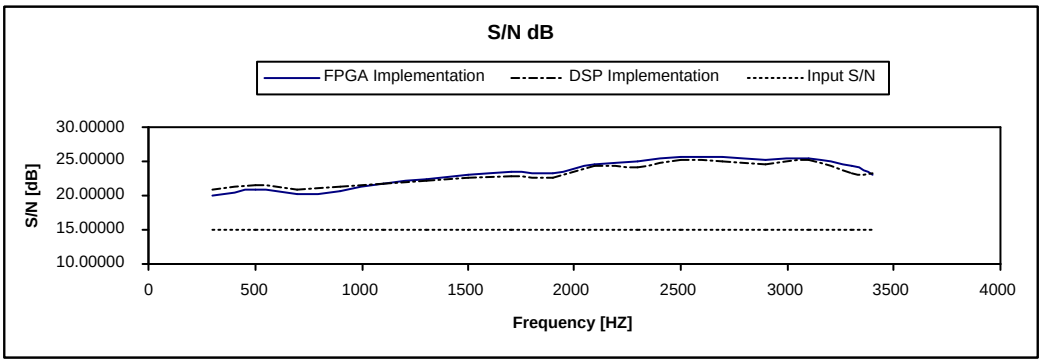

Figure 5.8: Comparison of the mixed demodulated signal With a SNR of 15dB at the input FM signal.

5.3.4 Processing Time Analysis

Processing speed is another index of the system performance. The time to measure here is time consumed in the whole process taken care by digital devices, which means the time from input of quadrature mixer to output of IIR filter.

The mixed demodulator architecture was modelled using VHDL (VHSIC Hardware Description Language) and synthesized with Synplcity Synplify pro 7.2 and mapped with Xilinx ISE 4 targeted at the Xilinx Virtex II XC2V500 FG264 FPGA device. The .srr file gives area-time analysis of the experimental application, which shows the process time as defined above is 15.62μs.

To fulfill the mixed demodulation functions, the implementation in FPGA take around 15.62μs. But the DSP processor need 32μs to finish the same function according to the Schnyder and Hallers report [39]. It shows that hardware implementation is almost two times faster than software solution in DSP processor.

5.4 Summary

This chapter showed test results on signal quality, robustness and processing time. Test results further verified the hardware solution.

Chapter 6

Conclusions

FPGA/ASIC is able to provide high-performance and flexible solutions to demanding signal processing applications. Currently most of FM demodulator is implemented based on software solution with DSP. A fully digitalized FM demodulator with catering to hardware architecture is main interest of this project.

A mixed FM demodulation algorithm, which is a combination of a delay demodulator and a phase adapter demodulator, is selected. It can overcome some disadvantages of other demodulation algorithms. For example, the phase-adapter demodulator only useful for narrow-band FM, the PLL has worse frequency response to chirp signal, etc.

CORDIC vector rotation is introduced to solve the trigonometric calculation and eliminate the division operations of the algorithm due to its iterative nature. Its architecture is also easily mapped to hardware architecture.

The mixed demodulator algorithm is finally implemented on a FPGA. The implementation has been tested for signal quality and robustness. Due to the hardware limitation currently, the hardware solution can work with up to 105 MHz modulated input signals. If higher frequency is required, down conversion should be added before connecting to the demodulator.

From the test results, the mixed demodulator implemented in FPGA can achieve almost same signal quality as in DSP processor. The robustness is a little worse than implementation in DSP processor. But the processing speed is two times faster. Obviously, fully digitalized mixed FM demodulator is a good choice for replacing the corresponding analog system, especially when the data need to be processed faster.

This project basically implemented the digitalized mixed FM demodulator. There are still some improvements to be done or research further in the future.

- Performance. To further improve the signal quality, the adaptive seeking of the carrier angular frequency in fixed-point is recommended to implemented.
- Process speed. With the technology develops, higher process speed is desired to the application.
- Power consumption is another point to improve further to meet the low power requirement like application supplied with battery. Especially with fast speed, the power will consume more.
- Area optimization. To implement the mixed FM demodulator in ASIC, some new algorithms can be applied such as LNS, even demodulator algorithm itself.

Bibliography

[1] R. Andraka, "A survey of cordic algorithms for fpga based computers," *FPGA 98 Monterey*, vol. 27, 1998.

[2] R. J. Andraka, "Building a high performance bit-serial processor in an fpga," in *Proc. of Design SuperCon 96*, pp. 5.1–5.21, 1996.

[3] Caffery, *ECECS 644: DSP Laboratory*. Lecture Notes, 2000.

[4] J. N. Coleman, *A High Speed Logarithmic Arithmetic Unit*. IESPRIT Long-term Research project No.33544 HSLA.

[5] J. N. Coleman and E. I. Chester, "A 32-bit logarithmic arithmetic unit and its performance compared to the floating point," *IEEE Trans. Computers*, vol. 49, pp. 702–715, July 2000.

[6] F. R. Connor, *Introductory Topics in Electronics and Telecommunication Modulation*. Edward Arnold, 1983.

[7] E. Deprettere, P. Dewilddle, and R. Udo, "Pipelined cordic architecture for fast vlsi filtering and array processing," in *Proc. ICASSP84*, pp. 41.A.6.1–41.A.6.4, 1984.

[8] C. Dick and F. J. Harris, "Configurable logic for digital communications: Some signal processing perspectives," *IEEE Communications Magazine*, pp. 107–111, August 1999.

[9] T. W. Dittmer, "Advances in digitally modulated rf systems," in *IEE International Broadcasting Convention conference Publication No.447*, pp. 427–431, 1997.

[10] W. J. Duh and J. M. Wu, "Implementing the discrete cosine transform by using cordic techniques," in *Proc. the International Symposium on VLSI Technology, Systems and Applications*, pp. 281–285, 1989.

[11] J. Dunning, "An all-digital phase-locked loop with 50-cycle lock time suitable for high-performance microprocessors," *IEEE JSSC*, vol. 30, pp. 412–422, April 1995.

[12] J. Duprat and J. M. Muller, "The cordic algorithm: New results for fast vlsi implementation," *IEEE transactions on communications*, vol. 42, pp. 168–178, 1993.

[13] K. Feher, *Digital Modulation Techniques in an Interference Environment*. Don White Consultants, Inc., 1977.

[14] S. Freeman and M. O'Donnell, "A complex arithmetic digital signal processor using cordic rotators," *Acoustics, Speech, and Signal Processing, ICASSP-95*, vol. 5, pp. 3191–3194, 1995.

[15] R. Fried, "Low-power digital pll with one cycle frequency lock-in time for clock synthesis up to 100mhz using 32,768hz reference clock," in *IEEE ASIC Conf. Proceedings*, pp. 291–294, 1996.

[16] J. J. Gibson and D. R. McClary, "A five-sample digital fm-demodulator with application to multichannel tc sound," *IEEE THAM 12.4*, pp. 164–165, 1988.

[17] E. Grayver and B. Daneshrad, "Direct digital frequency synthesis using a modified cordic," *IEEE transactions on communications*, pp. 241–244, 1998.

[18] G. J. Hekstra and E. F. A. Deprettere, "Floating point cordic," in *Proc. of the 11th symposium on Computer Arithmetic*, 1993.

[19] Y. H. Hu and S. Naganathan, "A novel implementation of chirp-z transformation using a cordic processor," *IEEE Transactions on ASSP*, vol. 38, pp. 352–354, 1990.

[20] ——, "An angle recoding method for cordic algorithm implementation," *IEEE Transactions on Computers*, vol. 42, pp. 99–102, January 1993.

[21] N. Isamailoglu and T. Yalcin, "Low-power design od a digital fm demodulator based on zero-cross detection at if," *IEEE*, pp. 810–813, 1999.

[22] H. T. Jensen and I. Galton, "A low-complexity dynamic element matching dac for direct digital synthesis," *IEEE Transactions on citcuits and systems-II: Anaglog and Digital Signal Porcessing*, vol. 45, pp. 13–27, January 1998.

[23] M. Johnson and E. Hudson, "A variable delay line phase locked loop for cpu-coprocessor synchronization," *IEEE J. Solid-State Circuits*, vol. 31, October 1988.

[24] W. Kester, *High Speed DACs And DDS Systems*. Analog Devices.

[25] N. G. Kingsbury and P. J. W. Rayner, "Digital filtering using logarithmic arithmetic," *Electronics Letters*, vol. 7, pp. 56–58, 1971.

[26] K. Konishi, H. Hitomi, H. Naka, K. Oishi, and M. Yamazaki, "An fm audio lsi for vhs vcrs using digital signal processing," *IEEE Transactions on Consumer Electronoics*, vol. 36, pp. 628–634, August 1990.

[27] T. Kyrokawa, J. A. Payne, and S. C. Lee, "Error analysis of recursive digital filters implemented with logarithmic number systems," in *IEEE Trans. Acoustics, Speech and Signal Processing*, pp. 706–715, 1980.

[28] T. Lee, "A 2.5 v cmos delay-locked loop for 18 mbit, 500 mb/s dram," *IEEE J. Solid-State Circuits*, vol. 29, December 1994.

[29] D. W. Lewis, "Interleaved memory function interpolators with application to an accurate lns arithmetic unit," *IEEE Trans. Computers*, vol. 43, pp. 974–982, 1994.

[30] ——, "114 mflops logarithmic number system arithmetic unit for dsp applications," *IEEE J. Solid-State Circuits*, vol. 30, pp. 1547–1553, 1995.

[31] J. Maneatis, "Low-jitter and process independent dll and pll based on self-biased techniques," in *ISSCC Digest of Technical Papers*, 1996.

[32] H. Meuth, A. Schnase, and H. Halling, "Possibilites and limitations for a fully-digital rf signal synthesis and control," *IEEE transactions on communications*, pp. 1253–1255, 1993.

[33] K. Miniami, "A 1ghz portable digital delay-locked loop with infinite phase capture ranges," in *ISSCC Digest of Technical Papers*, pp. 350–351, 2000.

[34] S. Mirabbasi and K. Martin, "Hierarchical qam: A spectrally efficient dc-free modulation scheme," *IEEE Communications Magazine*, pp. 140–146, November 2000.

[35] I. P. Nohuchi, Y. Daido, and J. A. Nossek, "Modulation technique for microwave digital radio," *IEEE Communications Magazine*, October 1986.

[36] V. Pasham, A. Miller, and K. Chapman, *Transposed Form FIR Filters*. Xilinx, 2001.

[37] P. Raha, "A robust digital delay line architecture in a 0.13 um cmos technology node for reduced design and process sensitivities," in *Proceedings of the International Symposium on Quality Electronic Design (ISQED02)*, 2002.

[38] T. Rahkonen, E. Malo, and J. Kostamovaara, "A 3v fully integrated digital fm demodulator based on a cmos pulse-shrinking delay line," *IEEE*, pp. 572–575, 1996.

[39] F. Schnyder and C. Haller, *Implementation of FM Demodulator Algorithms on a High Performance Digital Signal Processor*. 2002.

[40] T. Shiato and T. Okada, "A high capacity fully digitalized modulator for digital radios," *IEEE Communications Magazine*, pp. 1870–1874, 1992.

[41] L. H. Sibul and A. L. Fogelsanger, "Application of coordinate rotation algorithm to singular value decomposition," *IEEE Int. Symp. Circuits and Systems*, pp. 821–824, 1984.

[42] J. Singh, *High Speed Analog-to-Digital Converter for Software Radio Applications*.

[43] B. Sklar, *Digital Communications Fundamentals And Applications*. Prentice-Hall, 1988.

[44] B. Song and I. S. Lee, "A digital fm demodulator for fm, tv and wireless," *IEEE Transactions on Circuits And Systems*, vol. 42, pp. 821–825, December 1995.

[45] R. Sorace, "Digital-to-rf conversion for a vector modulator," *IEEE transactions on communications*, vol. 48, pp. 540–542, April 2000.

[46] E. E. Swartzlander and A. G. Alexopoulos, "The sign logarithm number system," *IEEE Trans. Computers*, pp. 1238–1242, 1975.

[47] E. E. Swartzlander, D. V. S. Chandra, H. T. Nagle, and S. A. Starks, "Logarithm arithmetic for fft implementation," *IEEE Trans. Computers*, vol. C-32, pp. 526–534, 1983.

[48] F. J. Taylor, R. Gill, J. Joseph, and J. Radke, "A 20 bit logarithmic number system processor," *IEEE Trans. Computers*, vol. 37, pp. 190–200, 1988.

[49] M. G. Thomas, T. A. bailey, J. R. Cowles, and M. D. Winkel, "Arithmetic cotransformations in the real and complex logarithmic number systems," *IEEE Trans. Computers*, vol. 47, pp. 777–786, 1998.

[50] J. Vankka, I. Sanchis, M. Kosunen, and K. Halonen, "A cordic based qam modulator," in *Global Telecommunications Conference*, pp. 300–303, 2000.

[51] J. E. Volder, "The cordic trigonometric computing technique," *IEEE transactions on electronic computers*, vol. 8, pp. 330–334, September 1959.

[52] J. S. Walther, "A unified algorithm for elementary functions," in *Proc. the AFIPS Spring Joint Computer Conference*, pp. 379–385, 1971.

[53] Xilinx, *Using the Virtex Delay-Locked Loop*. Xilinx, 2003.

[54] L. K. Yu and D. W. Lewis, "A 30-b integrated logarithmic number system processor," *IEEE J. Solid-State Circuits*, vol. 26, pp. 1433–1440, 1991.

Author's Publications

Conference

[1] F.B. Yu, "FPGA Implementation of a Fully Digital FM Demodulator", *Ninth IEEE International Conference on Communication Systems (ICCS 2004)*, (Singapore), pp. 446-451, September 2004.

Appendix A

Tools

This appends the lists of tools used in this project.

Software Development Tools

- Synplify Pro V7.2
- Xinlinx ISE 4

Hardware Development Tools

- Xilinx FPGA Proto Board Virtex-II
- Analog-to-Digital Converter THS1408EVM

Signal And Measurement Instruments

- Signal Generator Agilent 33250A
- Oscilloscope Tektronix TDS3054
- Sound Analyzer NCI A2-D

Appendix B

Measurement Data

This appends raw measurement data for this project.

Table B.1: SINAD of mixed demodulator

Message signal frequency [HZ]	FPGA implementation
200	-34.1
300	-34.6
500	-35.2
700	-34.5
900	-33.3
1100	-33.1
1300	-35.9
1500	-35.7
1700	-36.1
1900	-36.0
2100	-36.2
2300	-36.1
2500	-35.6
2700	-35.9
2900	-35.4
3100	-34.9
3300	-33.9
3400	-32.8
3500	-30.2
3700	-17.9

Table B.2: Mixed demodulator with -20 dB noise input

Message signal frequency [HZ]	FPGA implementation
300	26.8
500	28.2
700	27.9
900	27.6
1100	27.8
1300	28.5
1500	28.6
1700	28.9
1900	29.2
2100	29.8
2300	30.1
2500	30.2
2700	30.6
2900	30.2
3100	29.5
3300	28.5
3400	26.9

Table B.3: Mixed demodulator with -15 dB noise input

Message signal frequency [HZ]	FPGA implementation
300	19.8
500	20.8
700	20.2
900	20.6
1100	21.8
1300	22.4
1500	23.2
1700	23.6
1900	23.3
2100	24.7
2300	25.0
2500	25.6
2700	25.8
2900	25.2
3100	25.5
3300	24.5
3400	23.1

Appendix C

Source Codes

This appends source codes for the project. It includes top level mixed demodulator and module level mixer and filters.

```
--##################################################################################
-- Mixer demodulator

-- Engineer:  Yu Fubing
--            Xilinx Applications
-- Date :     6/2003

-- The following code implements a mixed demodulator.

--##################################################################################

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;

entity demodulater is
port ( RESET: in std_logic;
       CLOCK: in std_logic;
       DATAIN: in std_logic_vector(15 downto 0); -- Real part
       DATAOUT: out std_logic_vector(9 downto 0));
end demodulater;


architecture RTL of demodulater is

type Tri_table is array (0 to 4) of std_logic_vector(15 downto 0);

constant sin_table: Tri_table := ("0000000000000000", "0111100110111100",
"0100101100111100", "1100101100111100", "1111100110111100");

constant cos_table: Tri_table := ("1111111111111111", "0010011110001110",
"1110011110001110", "1110011110001110", "0010011110001110");


signal lockre, lockim: std_logic;
signal DATA_L: std_logic_vector(15 downto 0);
signal resin, imcos: std_logic_vector(31 downto 0);
signal FRE_IN, FIM_IN: std_logic_vector(15 downto 0);
signal remix, immix: std_logic_vector(15 downto 0);
signal MRE_IN, MIM_IN: std_logic_vector(15 downto 0);

component mixer
    port (CLOCK : in std_logic;
          RESET : in  std_logic;
          INRE: in std_logic_vector(15 downto 0); -- Real part
          INIM: in std_logic_vector(15 downto 0); -- Imaginal part
          MIXER_OUT: out std_logic_vector(9 downto 0));
end component;

component Filter
      port
          (
          RESET: in std_logic;
        CLOCK: in std_logic;
        INA: in std_logic_vector(15 downto 0);
        LOCKED: out std_logic;
```

```
        FIR_OUT: out std_logic_vector(15 downto 0)
          );
end component;

begin

process(CLOCK, RESET, DATAIN)
   variable i: integer range 0 to 5;
   begin

     if(RESET = '1') then
         DATA_L <= (others =>'0') ;
       FRE_IN <= (others => '0');
         FIM_IN <= (others =>'0') ;
       resin <= (others =>'0') ;
         imcos <= (others =>'0') ;
         i := 0;

     elsif(CLOCK'event and CLOCK ='1') then
          DATA_L <= DATAIN;
            resin <= DATA_L * sin_table(i);
            imcos <= DATA_L * cos_table(i);
            FRE_IN <= resin(31 downto 16);
            FIM_IN <= imcos(31 downto 16);
            i := i+1;
            if (i = 5) then
               i := 0;
            end if;
         end if;
end process;

REIIRFilter: Filter
      port map
          (
          RESET => RESET,
        CLOCK => CLOCK,
        INA => FRE_IN,
        LOCKED => lockre,
        FIR_OUT => MRE_IN
          );


IMIIRFilter: Filter
      port map
          (
          RESET => RESET,
        CLOCK => CLOCK,
        INA => FIM_IN,
        LOCKED => lockim,
        FIR_OUT => MIM_IN
          );

process(CLOCK, RESET, MRE_IN, MIM_IN)
   begin

     if(CLOCK'event and CLOCK ='1') then
          remix <= MRE_IN;
```

```
            immix <= MIM_IN;
        end if;
end process;

DemodulatorMixer: mixer
    port map
          (CLOCK => CLOCK,
          RESET => RESET,
          INRE => remix,
          INIM => immix,
          MIXER_OUT => DATAOUT);

end RTL;

--################################################################################
-- Mixer demodulator

-- Engineer:  Yu Fubing
--            Xilinx Applications
-- Date :     6/2003

-- The following code implements a mixed demodulator.

--################################################################################

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;

entity mixer is
port ( RESET: in std_logic;
       CLOCK: in std_logic;
       INRE: in std_logic_vector(15 downto 0); -- Real part
       INIM: in std_logic_vector(15 downto 0); -- Imaginal part
       MIXER_OUT: out std_logic_vector(9 downto 0));
end mixer;


architecture RTL of mixer is

signal CLK, CLKx4: std_logic;
signal DLL_LOCK: std_logic;
signal RE1, RE0, IM1, IM0: std_logic_vector(15 downto 0);
signal RE_OUT, IM_OUT: std_logic_vector(10 downto 0);
signal mul0, mul1: std_logic_vector(31 downto 0);
signal sIterationCount  : std_logic_vector(3 downto 0);
signal sStartVector     : std_logic;

component DLL_CLOCK
    port (CLKIN : in std_logic;
          RESET : in  std_logic;
          CLK0  : out std_logic;
          CLKDV : out std_logic;
          LOCKED: out std_logic);
end component;
```

```
component ntyDivideByTenState -- 10 times iteration
      port
          (
          pClock  : in std_logic;
          pStart  : in std_logic;
          pState  : out std_logic_vector (3 downto 0)
          );
end component;

component ntyVectorCordic
  port
    (
    pClock           : in std_logic;
      pStart                  : in std_logic;
    pControllerState : in std_logic_vector ( 3 downto 0);
    pInpX            : in  std_logic_vector(8 downto 0);
    pInpY            : in  std_logic_vector(8 downto 0);
    pArg             : out std_logic_vector(9 downto 0)
    );
end component;


begin

DLLmap: DLL_CLOCK port map(CLKIN => CLOCK, RESET => RESET, CLK0 => CLKx4,
                     CLKDV=> CLK, LOCKED => DLL_LOCK);

process(CLOCK, RESET, INIM, INRE)
   variable count: integer range 0 to 3;
   begin

     if(RESET = '1') then
      --   MIXER_OUT <= (others =>'0') ;
       RE0 <= (others => '0');
         RE1 <= (others => '0');
       IM0 <= (others =>'0') ;
       IM1 <= (others =>'0') ;
       RE_OUT <= (others =>'0') ;
       IM_OUT <= (others =>'0') ;
       mul0 <= (others =>'0') ;
       mul1 <= (others =>'0') ;
     --  sIterationCount <= (others =>'0') ;
         sStartVector <='0';

     elsif(CLOCK'event and CLOCK ='1') then
            sStartVector <= '1';
       count := count +1;
         if(count = 0) then
            RE0 <= INRE;
              IM0 <= INIM;
            mul0 <= RE1 * IM0 - IM1 * RE0;
                mul1 <= IM1 * IM0 + RE1 * RE0;
                RE_OUT <= mul0(31 downto 21);
                IM_OUT <= mul1(31 downto 21);
                RE1 <= RE0;
                IM1 <= IM0;
         end if;
```

```
     end if;
 end process;

TenIteration: ntyDivideByTenState
      port map
      (
      pClock => CLKx4,
      pStart => sStartVector,
      pState => sIterationCount
      );
VectorCordicmap: ntyVectorCordic port map(pClock => CLKx4, pStart =>
sStartVector, pControllerState =>sIterationCount, pInpX => RE_OUT,
pInpY => IM_OUT, pArg => MIXER_OUT);
end RTL;

--################################################################################
-- Xilinx DLL

-- The following code implements a clock DLL to generate a divide by four
-- operation.
-- One DLL is used to generate clock divide operation

--################################################################################



library ieee;
use ieee.std_logic_1164.all;



-- Functional Simulation Libraries
-- PRAGMA translate_off
--library unisim;
--use unisim.vcomponents.all;
-- PRAGMA translate_on



-- The folowing library declarartions are specific to Synplify synthesis
-- tool.

library virtex;
use virtex.components.all;

entity DLL_CLOCK is
    port (CLKIN : in  std_logic;
          RESET : in  std_logic;
          CLK0  : out std_logic;
          CLKDV: out std_logic;
          LOCKED: out std_logic);
end DLL_CLOCK;

architecture structural of DLL_CLOCK is

signal CLKIN_w, RESET_w, CLK0_dll, CLKDV_dll, CLKDV_g, CLK0_g,
LOCKED_dll,CLK0_w : std_logic;
```

```
component IBUFG
port(
      O   : out   STD_LOGIC;
      I   : in    STD_LOGIC);
end component;


component BUFG
   port(
      O  :  out   STD_LOGIC;
      I  :  in    STD_LOGIC);
end component;


component CLKDLL
    port ( CLKIN   : in  std_logic := '0';
           CLKFB   : in  std_logic := '0';
           RST     : in  std_logic := '0';
           CLK0    : out std_logic := '0';
           CLK90   : out std_logic := '0';
           CLK180  : out std_logic := '0';
           CLK270  : out std_logic := '0';
           CLK2X   : out std_logic := '0';
           CLKDV   : out std_logic := '0';
           LOCKED  : out std_logic := '0');
end component;



begin

clkpad : IBUFG  port map (I=>CLKIN, O=>CLKIN_w);


dll    : CLKDLL
             port map (CLKIN=>CLKIN_w, CLKFB=>CLK0_g, RST=>RESET,
                          CLK0=>CLK0_w, CLK90=>open, CLK180=>open, CLK270=>open,
                          CLK2X=>CLKDV_dll,    CLKDV=>open, LOCKED=>LOCKED);

clkg   : BUFG   port map (I=>CLK0_w,   O=>CLK0_g);
clkdvg : BUFG   port map (I=>CLKDV_dll,   O=>CLKDV_g);


CLK0 <= CLK0_g;
CLKDV <= CLKDV_g;

end structural;

--------------------------------------------------------------------------------
-- Name        : ntyDivideByTenState.vhd
-- Description : This module divides the main clock by 8 to generate eight
--               states for the pipelined modules of the Correction Unit.
--
--------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
```

```
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

--use work.pkgCore.all;

entity ntyDivideByTenState is
  port (
    pClock : in  std_logic;
    pStart : in std_logic;
    pState : out std_logic_vector (3 downto 0)
  );
end ntyDivideByTenState;

architecture arcDividebyTenState of ntyDivideByTenState is

begin
  lblState : process (pClock, pStart)
    variable vCount : std_logic_vector (3 downto 0);
  begin
    if rising_edge (pClock) and pStart = '1' then
      case vCount is
        when "0000" =>
          vCount := "0001";
        when "0001" =>
          vCount := "0010";
        when "0010" =>
          vCount := "0011";
        when "0011" =>
          vCount := "0100";
        when "0100" =>
          vCount := "0101";
        when "0101" =>
          vCount := "0110";
        when "0110" =>
          vCount := "0111";
        when "0111" =>
          vCount := "1000";
        when "1000" =>
          vCount := "1001";
        when "1001" =>
          vCount := "0000";
        when others =>
          vCount := "0000";
      end case;
      pState <= vCount;
    end if;
  end process lblState;

end arcDivideByTenState;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.all;

entity ntyVectorCordic is
  port
```

```
    (
    --Control
    pClock           : in std_logic;
    pStart             : in std_logic;
    pControllerState : in std_logic_vector ( 3 downto 0);

    -- Input
    pInpX            : in  std_logic_vector(10 downto 0);
    pInpY            : in  std_logic_vector(10 downto 0);

    --  OutPut, is unscale,
    -- must be divide by 1.16443534 to get the correct value
    -- output, 9 bit int, 8 bit fraction
    -- pMag             : out std_logic_vector(16 downto 0);

    -- ang is 10 bit fraction
    pArg             : out std_logic_vector(9 downto 0)
    );
end ntyVectorCordic;

architecture arcVectorCordic of ntyVectorCordic is

---------------------Component declaration -------------------------------
  Component ntyVectorCordicZConstandRegister
    port
      (
      pClock             : in std_logic;
      pStart             : in std_logic;
      pControllerState   : in std_logic_vector(3 downto 0);
      pZConstand         : out std_logic_vector(13 downto 0)
      );
  end Component;

  Component ntyVectorCordicIteration
    port
      (
      px0               : in std_logic_vector (23 downto 0);
      pxs0              : in std_logic_vector (21 downto 0);
      py0               : in std_logic_vector (23 downto 0);
      pys0              : in std_logic_vector (21 downto 0);
      pz0               : in std_logic_vector (15 downto 0);
      pz_const                 : in std_logic_vector (13 downto 0);
      pzout                   : out std_logic_vector (15 downto 0);
      pxout                   : out std_logic_vector (23 downto 0);
      pyout             : out std_logic_vector (23 downto 0)
      );
  end component;

  Component ntyVectorCordicShifter
    port
      (
      -- Control
      pState                 : in  std_logic_vector ( 3 downto 0);
      pClock                 : in std_logic;
      pStart                      : in std_logic;

      -- Internal fedBack input
```

```
      pInpX          : in std_logic_vector (23 downto 0);
      pInpY          : in std_logic_vector (23 downto 0);

      -- External Input
      pXmain         : in  std_logic_vector(10 downto 0);
      pYmain         : in  std_logic_vector(10 downto 0);

      pXShifted      : out std_logic_vector(21 downto 0);
      pYShifted      : out std_logic_vector(21 downto 0)
      );
  end Component;

---------------------Component declaration End--------------------------------

  -- Signal from zConstandRegister
  Signal sZConstandRegisterZOut  : std_logic_vector ( 13 downto 0);

  -- Signal from XYShifter
  Signal sXYShifterXOut          : std_logic_vector ( 21 downto 0);
  Signal sXYShifterYOut          : std_logic_vector ( 21 downto 0);

  -- Signal From XYZRegister
  signal sXYZRegisterXOut        : std_logic_vector ( 23 downto 0);
  signal sXYZRegisterYOut        : std_logic_vector ( 23 downto 0);
  signal sXYZRegisterZOut        : std_logic_vector ( 15 downto 0);

  -- Signal From Vector Cordic Iteration
  signal sVectorCordicIterationXOut : std_logic_vector ( 23 downto 0);
  signal sVectorCordicIterationYOut : std_logic_vector ( 23 downto 0);
  signal sVectorCordicIterationZOut : std_logic_vector ( 15 downto 0);

  signal pMag             :  std_logic_vector(16 downto 0);
begin

  ZConstandRegister : ntyVectorCordicZConstandRegister
    port map
      (
      pClock           => pClock,
      pStart                  => pStart,
      pControllerState => pControllerState,
      pZConstand       => sZConstandRegisterZOut
      );

  XYShifter : ntyVectorCordicShifter
    port map
      (
      -- Control
      pState                => pControllerState,
      pClock                => pClock,
      pStart                     => pStart,
      -- Internal fedBack input
      pInpX                 => sVectorCordicIterationXOut,
      pInpY                 => sVectorCordicIterationYOut,

      -- External Input
      pXmain                => pInpX,
      pYmain                => pInpY,
```

```
    pXShifted              => sXYShifterXOut,
    pYShifted              => sXYShifterYOut

    );

-- process to Mulplexer External Input or Internal FeedBack
XYZRegister : process (pClock,pControllerState, pStart,
                       pInpY,pInpX,
                       sVectorCordicIterationXOut,
                       sVectorCordicIterationYOut,
                       sVectorCordicIterationZOut)
  variable vPreviousYIn   : std_logic;
  begin
    if rising_edge(pClock) and pStart = '1' then

        case pControllerState is
          when "0000" =>
            sXYZRegisterXOut <= "00" & pInpX & "000000" & "00000";
            sXYZRegisterYOut <= "00" & pInpY & "000000" & "00000";
            sXYZRegisterZOut <= (others=>'0');

            if vPreviousYIn /= '0' then
              pArg <= sVectorCordicIterationZOut(14 downto 5);
            else
              -- input y is 0
              pArg <= (others=>'0');
            end if;

            pMag <= sVectorCordicIterationXOut(21 downto 5);

            if pInpY /= 0 then
              vPreviousYIn := '1';
            else
              vPreviousYIn := '0';
            end if;

          when others =>
            sXYZRegisterXOut <= sVectorCordicIterationXOut;
            sXYZRegisterYOut <= sVectorCordicIterationYOut;
            sXYZRegisterZOut <= sVectorCordicIterationZOut;
        end case;

    end if;
  end process XYZRegister;

VectorCordicIteration : ntyVectorCordicIteration
  port map
    (
    px0     => sXYZRegisterXOut,
    pxs0    => sXYShifterXOut,
    py0     => sXYZRegisterYOut,
    pys0    => sXYShifterYOut,
    pz0     => sXYZRegisterZout,
    pz_const    => sZConstandRegisterZOut,
    pzout       => sVectorCordicIterationZOut,
    pxout       => sVectorCordicIterationXOut,
```

```
      pyout   => sVectorCordicIterationYOut
      );

end arcVectorCordic;

--#################################################################################
-- Transposed Form IIR Filters

-- Engineer:  Yu Fubing
--            Xilinx Applications
-- Date :     6/2003

-- The following code implements a full precision Transposed Form Filter with
-- 3 cascade 8 taps, 16 bit inputs and 14-bit signed coefficients.

-- Signal COEFF_SIGN [15:0] is assigned based on the signs of the
-- 8 tap Coeffiecients.
-- COEFF_SIGN[15] indicates sign of third cascade feedback tap 2 coefficient
-- COEFF_SIGN[14] indicates sign of third cascade feedback tap 1 coefficient
-- COEFF_SIGN[13] indicates sign of third cascade feedback tap 0 coefficient
-- COEFF_SIGN[12] indicates sign of third cascade forward tap 2 coefficient
-- COEFF_SIGN[11] indicates sign of third cascade forward tap 1 coefficient
-- COEFF_SIGN[10] indicates sign of third cascade forward tap 0 coefficient
-- COEFF_SIGN[9] indicates sign of second cascade feedback tap 2 coefficient
-- COEFF_SIGN[8] indicates sign of second acscade feedback tap 1 coefficient
-- COEFF_SIGN[7] indicates sign of second cascade feedback tap 0 coefficient
-- COEFF_SIGN[6] indicates sign of second cascade forward tap 2 coefficient
-- COEFF_SIGN[5] indicates sign of second cascade forward tap 1 coefficient
-- COEFF_SIGN[4] indicates sign of second casacde forward tap 0 coefficient
-- COEFF_SIGN[3] indicates sign of first cascade feedback tap 1 coefficient
-- COEFF_SIGN[2] indicates sign of first cascade feedback tap 0 coefficient
-- COEFF_SIGN[1] indicates sign of first cascade forward tap 1 coefficient
-- COEFF_SIGN[0] indicates sign of first casacde forward tap 0 coefficient


--#################################################################################




library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
--use work.pack_rom16x18.all;

entity FILTER is
generic(tap_size: integer := 3);
port ( RESET: in std_logic;
       CLOCK: in std_logic;
       INA: in std_logic_vector(15 downto 0);
       LOCKED: out std_logic;
       FIR_OUT: out std_logic_vector(15 downto 0));
end FILTER;
```

```
architecture RTL of FILTER is
type bus30 is array (0 to tap_size-1) of std_logic_vector(29 downto 0);
type bus30_f is array (0 to 1) of std_logic_vector(29 downto 0);
signal COEFF_SIGN :std_logic_vector(15 downto 0);
signal CLK, CLKx4: std_logic;
signal DLL_LOCK: std_logic;
signal A_REG,A_REG_C1, A_REG_C2: std_logic_vector(15 downto 0);
signal A_REG0, A_REG1, A_REG2: std_logic_vector(15 downto 0);
signal mul0, mul1: std_logic_vector(29 downto 0);   --first cascade forward
signal mul2, mul3: std_logic_vector(29 downto 0);   --first cascade feedback
signal mul4, mul5, mul6: std_logic_vector(29 downto 0);  -- second cascade
forward
signal mul7, mul8, mul9: std_logic_vector(29 downto 0);  --second cascade
feedback
signal mul10, mul11, mul12: std_logic_vector(29 downto 0);   --third cascade
forward
signal mul13, mul14, mul15: std_logic_vector(29 downto 0);   --third cascade
feedback

signal sum1: std_logic_vector(29 downto 0);
signal sum0: std_logic_vector(30 downto 0); --first cascade forward
signal sum3: std_logic_vector(29 downto 0);
signal sum2: std_logic_vector(30 downto 0); --first cascade feedback

signal sum6: std_logic_vector(29 downto 0);
signal sum5: std_logic_vector(30 downto 0);
signal sum4: std_logic_vector(31 downto 0);     --second cascade forward

signal sum9: std_logic_vector(29 downto 0);
signal sum8: std_logic_vector(30 downto 0);
signal sum7: std_logic_vector(31 downto 0); --second cascade feedback

signal sum12: std_logic_vector(29 downto 0);
signal sum11: std_logic_vector(30 downto 0);
signal sum10: std_logic_vector(31 downto 0);--third cascade forward
signal sum15: std_logic_vector(29 downto 0);
signal sum14: std_logic_vector(30 downto 0);
signal sum13: std_logic_vector(31 downto 0); --third cascade feedback

signal sign_in: std_logic;
signal kcm_result0, kcm_result_f0: bus30_f;
signal kcm_result1, kcm_result2, kcm_result_f1, kcm_result_f2: bus30;


component DLL_CLOCK
    port (CLKIN : in std_logic;
          RESET : in  std_logic;
          CLK0  : out std_logic;
          CLKDV : out std_logic;
          LOCKED: out std_logic);
end component;

component KCM
generic(select_rom: integer);
port(RESET        : in std_logic;
       CLK        : in std_logic;
     CLKx4        : in std_logic;
```

```
        ENABLE    : in std_logic;
       SIGN_IN    : in std_logic;
       SIGN_COEFF : in std_logic;
       Digital_in : in std_logic_vector(15 downto 0);
     MUL_OUT      : out std_logic_vector(29 downto 0));
end component;


begin

COEFF_SIGN <= "0000110000110011";

DLLmap: DLL_CLOCK port map(CLKIN => CLOCK, RESET => RESET, CLK0 => CLKx4,
                     CLKDV=> CLK, LOCKED => DLL_LOCK);

-- Instantiate eight different KCMs with different ROMs.

KCM_MULTIPLIERs_Cas1For: for tap in 0 to 1 generate


U1:   KCM   generic map(select_rom => tap)
                   port map(CLK => CLK, CLKx4 => CLKx4, RESET => reset, ENABLE
=> DLL_LOCK,
                      SIGN_IN => sign_in, SIGN_COEFF => COEFF_SIGN(tap),
                   Digital_in => A_REG0, MUL_OUT => kcm_result0(tap));
                 end generate;

KCM_MULTIPLIERs_Cas1Feed: for tap in 0 to 1 generate
U2:   KCM   generic map(select_rom => (tap+2))
                   port map(CLK => CLK, CLKx4 => CLKx4, RESET => reset, ENABLE
=> DLL_LOCK,
                      SIGN_IN => sign_in, SIGN_COEFF => COEFF_SIGN(tap+2),
                   Digital_in => A_REG0, MUL_OUT => kcm_result_f0(tap));
                 end generate;

KCM_MULTIPLIERs_Cas2For: for tap in 0 to 2 generate
U3:   KCM   generic map(select_rom => (tap+4))
                   port map(CLK => CLK, CLKx4 => CLKx4, RESET => reset, ENABLE
=> DLL_LOCK,
                      SIGN_IN => sign_in, SIGN_COEFF => COEFF_SIGN(tap+4),
                   Digital_in => A_REG1, MUL_OUT => kcm_result1(tap));
                 end generate;

KCM_MULTIPLIERs_Cas2Feed: for tap in 0 to 2 generate
U4:   KCM   generic map(select_rom => (tap+7))
                   port map(CLK => CLK, CLKx4 => CLKx4, RESET => reset, ENABLE
=> DLL_LOCK,
                      SIGN_IN => sign_in, SIGN_COEFF => COEFF_SIGN(tap+7),
                   Digital_in => A_REG1, MUL_OUT => kcm_result_f1(tap));
                 end generate;

KCM_MULTIPLIERs_Cas3For: for tap in 0 to 2 generate
U5:   KCM   generic map(select_rom => (tap + 10))
                   port map(CLK => CLK, CLKx4 => CLKx4, RESET => reset, ENABLE
=> DLL_LOCK,
                      SIGN_IN => sign_in, SIGN_COEFF => COEFF_SIGN(tap+10),
                   Digital_in => A_REG2, MUL_OUT => kcm_result2(tap));
```

```
                end generate;

KCM_MULTIPLIERs_Cas3Feed: for tap in 0 to 2 generate
U6:   KCM   generic map(select_rom => (tap + 13))
                  port map(CLK => CLK, CLKx4 => CLKx4, RESET => reset, ENABLE
=> DLL_LOCK,
                     SIGN_IN => sign_in, SIGN_COEFF => COEFF_SIGN(tap+13),
                  Digital_in => A_REG2, MUL_OUT => kcm_result_f2(tap));
                end generate;
LOCKED <= DLL_LOCK;

mul3 <= kcm_result_f0(1);
mul2 <= kcm_result_f0(0);
mul1 <= kcm_result0(1);
mul0 <= kcm_result0(0);

mul9 <= kcm_result_f1(2);
mul8 <= kcm_result_f1(1);
mul7 <= kcm_result_f1(0);
mul6 <= kcm_result1(2);
mul5 <= kcm_result1(1);
mul4 <= kcm_result1(0);

mul12 <= kcm_result_f2(2);
mul11 <= kcm_result_f2(1);
mul10 <= kcm_result_f2(0);
mul15 <= kcm_result2(2);
mul14 <= kcm_result2(1);
mul13 <= kcm_result2(0);

process(CLK, RESET)
   begin

     if(RESET = '1') then
         sum0 <= (others => '0');
       sum1 <= (others =>'0') ;
       sum2 <= (others =>'0') ;
       sum3 <= (others =>'0') ;

         sum4 <= (others =>'0') ;
       sum5 <= (others =>'0') ;
       sum6 <= (others =>'0') ;
       sum7 <= (others =>'0') ;
         sum8 <= (others =>'0') ;
       sum9 <= (others =>'0') ;

         sum10 <= (others =>'0') ;
       sum11 <= (others =>'0') ;
       sum12 <= (others =>'0') ;
       sum13 <= (others =>'0') ;
         sum14 <= (others =>'0') ;
       sum15 <= (others =>'0') ;

         A_REG_C2 <=(others => '0');
         A_REG_C1 <=(others => '0');
         A_REG <= (others =>'0');
       A_REG0 <=(others => '0');
```

```
        A_REG1 <=(others => '0');
        A_REG2 <=(others => '0');
        FIR_OUT <=(others => '0');
        sign_in <= '0';
    elsif(CLK'event and CLK ='1') then
        A_REG <= INA - sum2(30 downto 15);
            A_REG_C1 <= (sum0 - sum7)(31 downto 16);
            A_REG_C2 <= (sum4 - sum13)(31 downto 16);
        if(A_REG(15) = '1') then
           A_REG0 <= (not A_REG) + '1';
             sign_in <= '1';
        else
           A_REG0 <=  A_REG;
             sign_in <= '0';
        end if;

      sum0  <=  (sum1(29) & sum1) + (mul0(29) & mul0);
      sum1  <=   mul1;

      sum2  <=  (sum3(29) & sum3) + (mul2(29) & mul2);
      sum3  <=   mul3;
       if(A_REG_C1(15) = '1') then
           A_REG1 <= (not A_REG_C1) + '1';
             sign_in <= '1';
        else
           A_REG1 <=  A_REG_C1;
             sign_in <= '0';
        end if;
      sum4  <=  (sum5(30) & sum5) + (mul4(29) & mul4(29) & mul4);
      sum5  <=  (sum6(29) & sum6) + (mul5(29) & mul5);
      sum6  <=   mul6;
      sum7  <=  (sum8(30) & sum8) + (mul7(29) & mul7(29) & mul7);
      sum8  <=  (sum9(29) & sum9) + (mul8(29) & mul8);
      sum9  <=   mul9;
       if(A_REG(15) = '1') then
           A_REG2 <= (not A_REG_C2) + '1';
             sign_in <= '1';
        else
           A_REG2 <=  A_REG_C2;
             sign_in <= '0';
        end if;
      sum10  <=  (sum11(30) & sum11) + (mul10(29) & mul10(29) & mul10);
      sum11  <=  (sum12(29) & sum12) + (mul11(29) & mul11);
      sum12  <=   mul12;
      sum13  <=  (sum14(30) & sum14) + (mul13(29) & mul13(29) & mul13);
      sum14  <=  (sum15(29) & sum15) + (mul14(29) & mul14);
      sum15  <=   mul15;
        FIR_OUT <= sum10(31 downto 16);
    end if;
 end process;
end RTL;
```