# Implementation and Applications of Logarithmic Signal Processing on an FPGA

A Thesis submitted to The University of Kent for the degree of Doctor of Philosophy in Electronic Engineering

By

Mandeep Chaudhary

February 2016

*Dedicated to my family, teachers & friends.*

# Acknowledgements

Firstly, I would like to thank my supervisor Dr. Peter Lee for his guidance and support. I thank him for supporting me with my PhD when I was having health problems. My meetings with him have been a source of great encouragement, inspiration and learning. Thank you for believing in me and giving me such a nice opportunity.

My friends and colleagues Dr. Harshal Oza, Gurtac Yemiscioglu, Mark Esdale and Steven J. Moser, thank you for keeping me smiling in difficult times. Thank you to the EDA staff for the opportunities and help you have given me for the years.

Finally I would like to say deepest thanks to my family in India. My father (Randhir Singh Malik) and mother (Kamla Devi) thank you for love and moral support. I would not have made this far without you all.

## Abstract

This thesis presents two novel algorithms for converting a normalised binary floating point number into a binary logarithmic number with the single-precision of a floating point number. The thesis highlights the importance of logarithmic number systems in real-time DSP applications. A real-time cross-correlation application where logarithmic signal processing is used to simplify the complex computation is presented.

The first algorithm presented in this thesis comprises two stages. A piecewise linear approximation to the original logarithmic curve is performed in the first stage and a scaled-down normalised error curve is stored in the second stage. The algorithm requires less than 20 kbits of ROM and a maximum of three small multipliers. The architecture is implemented on Xilinx's Spartan3 and Spartan6 FPGA family. Synthesis results confirm that the algorithm operates at a frequency of 42.3 MHz on a Spartan3 device and 127.8 MHz on a Spartan6. Both solutions have a pipeline latency of two clocks. The operating speed increases to 71.4 MHz and 160 MHz respectively when the pipeline latencies increase to eight clocks.

The proposed algorithm is further improved by using a PWL (Piece-Wise Linear) approximation of the transform curve combined with a PWL approximation of a scaled version of the normalized segment error. A hardware approach for reducing the memory with additional XOR gates in the second stage is also presented. The architecture presented uses just one 18k bit Block RAM (BRAM) and synthesis results indicate operating frequencies of 93 and 110 MHz when implemented on the Xilinx Spartan3 and Spartan6 devices respectively.

Finally a novel prototype of an FPGA-based four channel correlation velocimetry system is presented. The system operates at a higher sampling frquency than previous published work and outputs the new result after every new sample it receives. The system works at a sampling frequency of 195.31 kHz and a sample resolution of 12 bits. The prototype system calculates a delay in a range of 0 to 2.6 ms with a resolution of 5.12 $\mu$s.

# Publications

1. Chaudhary, M.; Lee, P, ”Two-stage logarithmic converter with reduced memory requirements,” *Computers & Digital Techniques, IET,* vol.8, no.1, pp.23,29, January 2014.

2. Chaudhary, M.; Lee, P, ”An Improved 2-Step Binary Logarithmic Converter for FPGAs,” *IEEE transactions on Circuits & Systems II : Express Briefing,* vol.62, no.5, pp.476,480, May 2015.

# Contents

# List of Figures

# List of Tables

# Abbreviations

| | | |
|---|---|---|
| BRAM | : | Block random access memory |
| DRAM | : | Distributed random access memory |
| DSP | : | Digital Signal Processing |
| DFT | : | Discrete Fourier Transform |
| FPGA | : | Field Programmable Gate Arrays |
| FFT | : | Fast Fourier Transform |
| GPU | : | Graphical processing unit |
| LSP | : | Logarithmic Signal Processing |
| LNS | : | Logarithmic Number System |
| LUT | : | Look Up Table |
| Log | : | Logarithm |
| LIN2LOG | : | Linear To Logarithm |
| LOG2LIN | : | Logarithm TO Linear |
| LOD | : | Leading One Detector |
| LSB | : | Least Significant Bit |
| LZD | : | Leading Zero Detector |
| MSB | : | Most Significant Bit |
| PWL | : | Piecewise Linear |
| PWP | : | Piecewise Polynomial |
| ROM | : | Read Only Memory |
| ULP | : | Unit Last Place |
| VLSI | : | Very Large Scale Integration |

# Chapter 1

# Introduction

## 1.1 Motivation, Aims & Objective

In the world today digital electronics are used in almost every other application. Whether, these applications are related to day-to-day activities, defence services or research work, they all perform computations in the digital field. The technologies used in implementing digital electronics in hardware have evolved with time. One of the important parameters for a computation is its precision (i.e. the bit resolution). The precision used in a computation is responsible for the result's accuracy.

There are many number systems used to perform computations in digital electronics. However, the two most common number systems used are the fixed and floating point number systems. These number systems are acceptable for low resolution simple arithmetic computations. The usage of fixed and floating point numbers in real time applications with higher bit resolution requires enormous and complex hardware architecture. This complex hardware architecture and real-time computation of data creates the problem of bottlenecks and inefficient systems for real-time computations. The research in this thesis solves the problem by using logarithm numbers instead of fixed/floating point numbers in computations. The research in this thesis designs a single floating point precision Lin2Log converter which uses less hardware (maintaining similar accuracy) than the recently published Lin2Log converters and uses it in a real-time application.

Logarithms were first introduced by John Napier in 1614. The logarithm of a number is defined as the exponent by which a number is raised to produce the same number again, for example

$$1000 = 10^3 \tag{1.1}$$

Thus, the number 1000 has a logarithm value of 3 with base 10. The logarithm mathematically can be expressed as

$$log_{10}(1000) = 3 \tag{1.2}$$

In computers/digital electronics a logarithm with base 2 otherwise known as binary logarithm is used. The binary logarithm is simple and easy to implement in hardware when compared to the implementation of other bases of logarithm in computers/digital electronics.

Logarithms have been used as a tool in mathematics, to simplify complex arithmetic operations. The logarithm properties of the linear operations of multiplication, division, power, addition and subtraction are shown in equations 1.3, 1.4, 1.5, 1.6 and 1.7, where $x$ and $y$ are the numbers in the linear domain. The use of logarithms reduces multiplication, division and power in the linear domain to addition, subtraction and multiplication in the log domain respectively.

$$log_b(x.y) = log_b(x) + log_b(y) \tag{1.3}$$

$$log_b(\frac{x}{y}) = log_b(x) - log_b(y) \tag{1.4}$$

$$log_b(x^p) = p.log_b(x) \tag{1.5}$$

$$log_b(x + y) = log_b|x| + log_b(1 + 2^{log_b|y|-1}) \tag{1.6}$$

$$log_b(x - y) = log_b|x| + log_b(1 - 2^{log_b|y|-1}) \tag{1.7}$$

The properties of logarithms save time for multiplication and division in a processor on a computer and reduce the area in integrated circuits by replacing hardware multipliers and shifters by adders and subtractors. Logarithmic signal processing has been used in a number of applications and is of increasing interest because of the potential for area and higher resolution architectures [1]. As a consequence, many algorithms and hardware architectures have been proposed over the past 50 years for converting a normalised binary number $x$ into a binary logarithm ($log_2 x$) or converting the binary exponent ($2^{log_2 x}$) back to a normalised binary number. These algorithms vary widely in terms of their accuracy, efficiency and speed. Over such a long time-span there have also been significant changes in the performance of the implementation technologies, resulting in newer architectures capable of achieving increasing conversion accuracies and making the logarithmic number system (LNS) a viable alternative in a number of modern applications [2–4], speech recognition [5] and digital hearing aids [6].

There are two generic architectures for LNS processing. The first architecture consists of all arithmetic operations ($\times, \div, +, -$) to be performed in the logarithmic domain (see figure 1.1), whereas in the second architecture, sometimes called hybrid-LNS, the arithmetic operations ($\times, \div$) are performed in the log domain while the remaining operations ($+, -$) are performed in the linear domain (see figure 1.2). The reason for using two different domains in hybrid-LNS is that logarithmic addition and subtraction are non-linear functions requiring further approximation methods (see equations 1.6 and 1.7), frequently based on piecewise linear (PWL) or higher-order piecewise polynomial (PWP) methods, thereby offsetting any advantages of using the LNS (Logarithmic Number System). The hybrid LNS architectures have been used in many applications such as image compression where the forward and inverse discrete cosine transforms (DCT) are performed on digital images [7].

The disadvantage of hybrid LNS is that for some algorithms repeated conversion to and from the log domain is necessary. The choice of architecture, the required

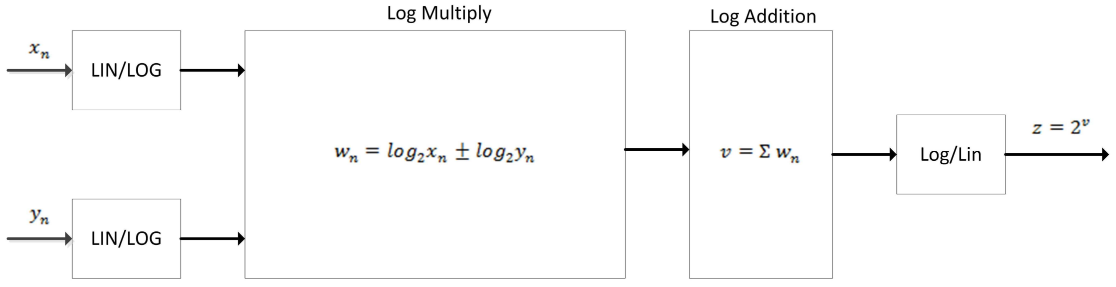


Figure 1.1: Standard logarithmic processor.

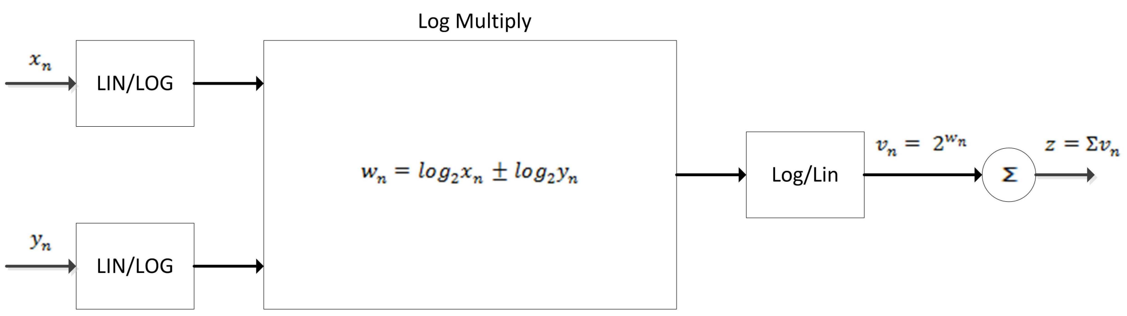


Figure 1.2: Hybrid logarithmic processor.

resolution and the complexity of the overall conversion algorithms are strongly application dependent and are only advantageous if the sum of complex arithmetic operations is reduced, resulting in decreased power consumption or reduced circuit area [8].

The recent research has tended to ignore a potential niche where logarithmic signal processing has advantages over both fixed and floating point solutions. Thus, the research work objective was to establish an algorithm for analysing and modelling the performance of a Lin2Log converter with single floating point precision [9]. As most of the LSP-based applications require a high dynamic range with limited accuracy, single floating point precision (i.e. 32 bit) is chosen for the purposes of this research.

Another objective of the research was to map the architecture of a logarithmic converter onto a reconfigurable logic device (i.e. FPGA). Finally in this research a cross-correlation-based application using the LNS instead of a fixed/floating point number system is presented to find the velocity of pneumatic particles flowing through a pipeline.

## 1.2 Research Contributions

The research work in this thesis provides information about the two novel algorithms [10, 11] for a Lin2Log converter and an application of cross-correlation using the LNS specifically designed for mapping onto a reconfigurable device. The research work here focuses on the area reduction, usage of minimum hardware and precision achieved when designing a new algorithm or implementing the LNS onto an application. In this thesis reconfigurable devices are chosen for hardware implementation as they can perform parallel computation processing at high frequencies in comparison to microprocessors and microcontrollers.

The thesis presents a novel algorithm [10] to convert a linear (fixed/floating point) number to an equivalent logarithmic number. The proposed algorithm [10] requires less than 20 kbits of ROM and a maximum of three small multipliers. The research shows how a generic error curve is generated in the first stage of the algorithm using max, mean and minimum error curves and then normalised for reproducing in the second stage. The thesis proposes another novel algorithm [11] to reduce the memory elements (in the second stage) used for storing the error curve of the first stage. The memory in the second stage is reduced further by 32% by exploiting the properties of symmetry in the normalised error curve. The algorithm [11] uses additional XOR gates for implementing reduced memory in the second stage.

A prototype velocimetry system used for real-time speed measurement of pneumatic particulates flowing through a pipeline using the LNS is presented. The system is developed using incremental cross-correlation in the time domain instead of using FFT (fast fourier transform) techniques in the frequency domain for the calculation of a continuous stream of data from multiple electrostatic sensors located in a pipeline. The system operates at a higher sampling frequency than in previously published work and outputs the new result after every new sample it receives. This thesis provides the results of implementing the circuit on an FPGA device and shows the reduction of bits in fractional bits (used for the linear to logarithmic converter).

The research work presented in this thesis has been submitted to and accepted by IET Computer and Digital Techniques [10] and IEEE Transactions on Circuit and Systems II [11].

## 1.3 Thesis Organisation

All the work presented in this thesis is organised in different chapters. The thesis begins with an overview of FPGA technology in Chapter 2. The generic FPGA architecture and features are compared with other alternative implementation technologies. The number representation using floating point and fixed point number systems are discussed. Published work on the implementation of floating point number systems on FPGA devices, for arithmetic operations is discussed and compared with logarithmic number systems.

Chapter 3 describes in detail the logarithmic number system and provides an overview of algorithms and techniques used for converting Lin2Log and vice versa. The popular algorithms proposed for normalising a binary number, using a leading one or leading zero detector, are explained in section 3.2 and domain conversion algorithms are explained in section 3.3. The algorithms described in Chapter 3 are re-simulated in MATLAB and are plotted with their individual accuracy achieved.

Chapter 4 begins with a description of the Lin2Log conversion algorithm proposed by K.E. Larson in 1994 [12]. The chapter proposes a novel algorithm performing improvements to Larson's algorithm [12]. The proposed novel algorithm is simulated using MATLAB and implemented onto newer and old families of FPGA devices. An analysis of different configurations of data bits, used for addressing and interpolation of the PWL (Piece-wise Linear) approximation and a detailed overview of the resolution of coefficient bits stored in memory to perform PWL approximation is provided in this chapter. The numerical data on precision achieved is compared with recent published papers.

Chapter 5 presents further improvement to the algorithm first proposed in Chapter 4. Chapter 5 describes a further improved algorithm, exploiting symmetrical properties of a normalised error curve. The implementation of a further improved algorithm on an FPGA device, by using dynamic RAM instead of Block RAM, is provided in the chapter. The algorithm is implemented in newer and old families of FPGA devices. An analysis of different configurations of data bits and resolution of coefficient bits stored in memory are provided in this chapter. The chapter compares the results obtained from the further improved algorithm with results obtained from

Chapter 4 and recent published papers.

Chapter 6 begins by describing correlation techniques and methods to speed up its computation such as using FFT (Fast Fourier Transform) and incremental correlation algorithms. The recent work used for velocimetry system is described in section 6.3. The velocimetry system implemented on an FPGA device in previous work is presented in section 6.4. A new algorithm for detecting the velocity of pneumatic particles flowing in a pipeline is mentioned in section 6.5. The results obtained from implementing the algorithm on an FPGA device and its utilisation of hardware resources are presented in section 6.7.

A review of all the research presented in this thesis, along with a summary of findings, is presented in Chapter 7 as a conclusion. Directions for further research are also suggested.

# Chapter 2

# FPGA Technology

## 2.1 Introduction

This Chapter contains a brief overview on FPGA (Field Programmable Gate Array) technology. The FPGA is a very flexible and powerful hardware, allowing implementation of large and complex logic designs. The FPGA generally uses a combination of gates and provides dedicated hardware resources to implement a real-time complex application.

The chapter begins with existing DSP system technologies for implementing a digital logic circuit in section 2.2. Section 2.3 starts with a brief history of FPGA technology. An overview on FPGA different programming technologies along with their main vendors are provided in section 2.4. An overview of low-cost families of FPGA devices, are provided in sections 2.4.1, 2.4.2 and 2.4.3. Section 2.5 provides a brief information on programming languages used for designing a digital circuit. Section 2.6 provides a description of a common value representation format implemented on FPGAs and their shortcomings.

## 2.2 DSP system Technologies

For implementing the DSP operations in the hardware a DSP system is required, the discrete signals are easily manipulated in a DSP system. The FPGA, ASIC, DSP, GPU and CPU are the few commonly used technologies for implementing a

DSP system in hardware.

ASICs (Application Specific Integrated Circuits) are ICs (integrated circuits) built only for a specific application. ASICs contains predefined/hardwired gates for specific applications because of which they do not offer flexibility of redesigning or reprogramming them. ASICs require a low level logic to design the device. The mask and design cost of ASICs in lower volume are higher than FPGA and DSP devices. However, when ASICs are made in high volume, the cost becomes more economical. It is been proved that for the same level of technology, ASICs are typically three to four times faster than the FPGA devices [13].

ASICs require refabrication if there are any errors found in the design. FPGAs were initially made to avoid the problem of refabrication of ASIC chips. The FPGAs are sea of gates present in hardware. A prototyping of the logic circuits is performed on FPGA. FPGA's reprogrammable feature makes it possible to make changes and run tests in the circuit before the circuit is send for fabrication. FPGA provides the feature of signal processing in parallel rather than sequence because of which multiple operations can be performed at the same time. Unlike ASICs, FPGAs require no layout, masks or other manufacturing steps in designing an integrated circuit. When implementing a system that requires reprogramming, parallel operations and cost effectiveness for lower volume production, FPGA technology is preferred over ASIC technology.

The above mentioned technologies were based on designing hardware circuit performing parallel operations. However, processors (CPU) used for simpler operations in general purpose computers can be used for performing DSP applications. A CPU uses a simple load-store program design. The CPUs have a fixed hardware structure which limits there memory, peripheral structures and connections in hardware. Due to which when performing a DSP application on general purpose processor a slower performance is obtained. To overcome the performance issues, the GPUs and DSPs were introduced. The GPUs are designed to accelerate creation of images for a computer display. A GPU consists of thousands of cores designed for handling multiple tasks simultaneously. The GPUs are designed to perform functions such as texture mapping, image rotation, translation, shading, etc. Modern GPUs are very efficient at manipulating computer graphics and image processing, and their highly parallel structure makes them more effective than general-purpose CPUs for algorithms where the processing of large blocks of visual data is done in parallel.

The DSPs are specialised integrated circuits for processing digital signals. They have better power consumption and are faster than general purpose processors. Fixed and floating point architectures are both present in DSPs. However, for speed efficiency fixed point binary architectures are preferred. The DSP architecture generally contains a program memory, data memory, ALU (arithmetic logic unit) and the input / output ports. The input / output ports connect to analogue signals by using an ADC (analogue to digital converter) and DAC (digital to analogue converter). In the architecture, the data memory stores the data to be processed and the program memory provides the data to ALU. In the ALU, instructions are executed sequentially. An example of fixed point digital signal processor is TMS320C6455. It comes in four packages : TMS320C6455-1000, TMS320C6455-1200, TMS320C6455-720 and TMS320C6455-850, each with 1 GHz, 1.2 GHz, 720 MHz and 850 MHz operating frequency respectively. This DSP comprises eight 32-bit instruction cycles and performs 9600 million instructions per second [14].

## 2.3 FPGA Overview

The FPGA was invented by Ross Freeman in 1984. The basic idea behind FPGA was to have a reprogrammable hardware, on which one or more particular application logic designs can be made. The instructions were executed in parallel on FPGA rather than in sequence, on conventional computers.

In the 1970s, programming logic devices were introduced. These devices were programmed either by fuse or masking logic. In the 1980s, EEPROMs (electrically erasable programmable read only memory) were used to control each programmable connection instead of a fuse. Another alternative method, PAL (programmable array logic), was made by fixing AND gates and making OR gates programmable. The structure resulted in a PROM (programmable read only memory) or LUT (look up table). The problem with PROM was that the circuit size grew exponentially with number of inputs [15]. Larger PROMs, were slower than the dedicated logic, and power consumption was higher, which restricted this technology.

In the mid 1980s programmable logic used several blocks of logic on a single chip to make complex logic functions. This advancement in technology resulted in CPLD (complex programmable logic devices) and FPGA (field programable gate arrays) devices. The CPLDS were based on PAL architecture, making them flash programmable. The FPGAs were based on LUT architecture, where the programme is held in static memory cells, making the configuration of FPGA devices volatile. With the development in technology, interconnection structure and availability of logic resources on FPGA devices increased. This made complex algorithms/architecures implementation easier and on a single device. In an FPGA architecture (shown in figure 2.1), there is a 2-D array layout of identical CLBs (Configurable Logic Blocks), input-output blocks and programmable interconnections.

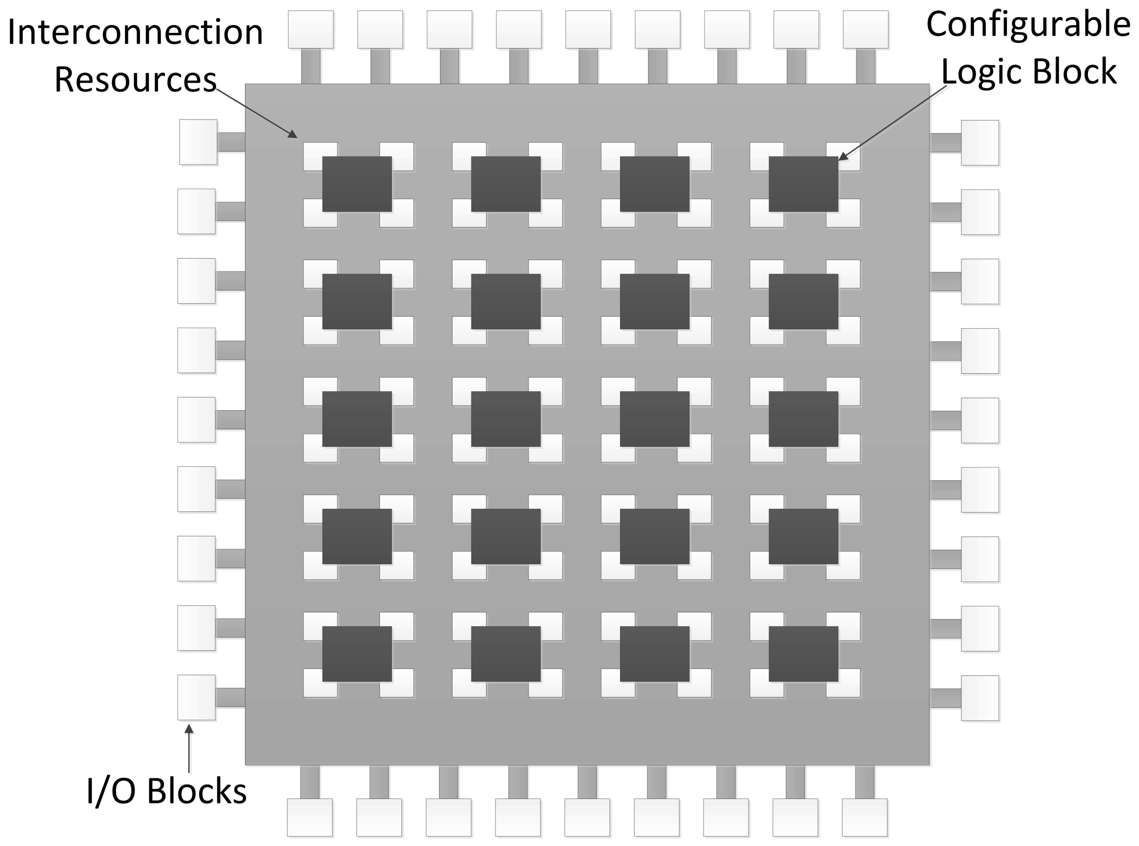


Figure 2.1: FPGA Architecture.

The input-output blocks connect the internal structure of FPGA to external devices. The CLBs are based on a LUT architecture and are interconnected through a programming routing matrix, enabling FPGA to achieve high flexibility in their designs. The CLBs in FPGA are made of 4 input LUT (see figure 2.2) or 6 input LUT (see figure 2.3) in modern FPGA devices, producing a single bit output. The flip-flop attached to the LUT is used to register the output. The complex functions in FPGA are implemented by cascading multiple LUTs.

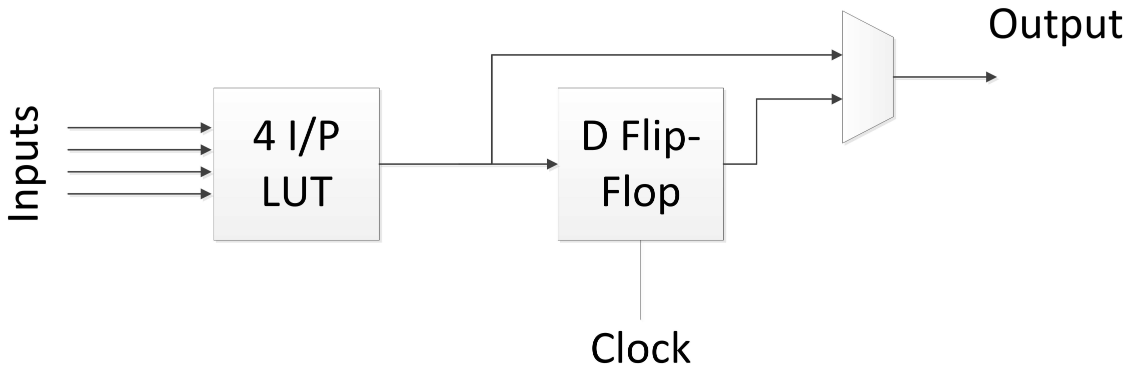


Figure 2.2: FPGA CLB using 4 input LUT.

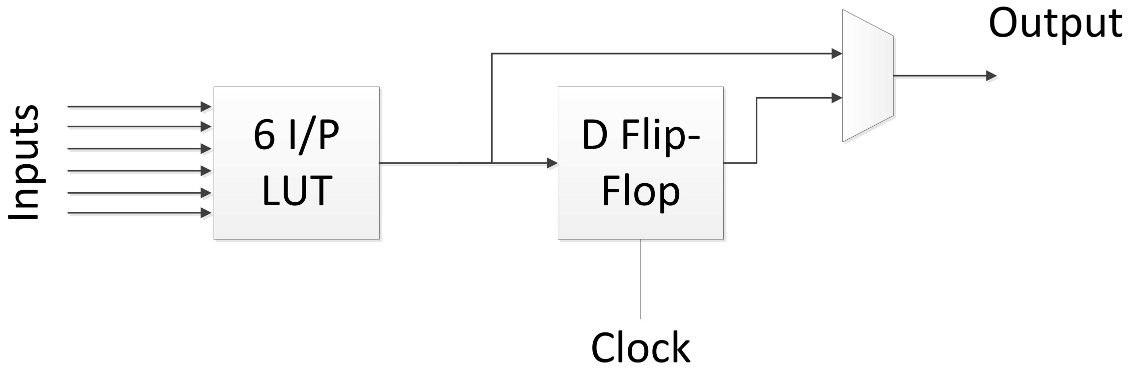


Figure 2.3: FPGA CLB using 6 input LUT.

It is not practically possible on the basis of speed and area to provide dedicated connection of every possible output to every possible input of CLBs. The solution of this problem was to have a set of routing lines, which can be shared to create connection between CLBs for a specific application. In an FPGA for grid interconnections a crossbar switch is used to programme connections between horizontal and vertical routing lines. In modern days due to the high input output technology, reconfigurable logic blocks with high flexibility and dedicated hardware resources in FPGAs are being used for implementing complete computer control systems such as the Commodore Amiga 500 in project MiniMig [16]. FPGAs have also been implemented to make a complete computer using linux [17–19].

## 2.4 FPGA Device Types and Families

In the current market, FPGA devices uses anti-fuse, flash and SRAM based programming technology. In the anti-fuse programming technology, the devices are configured by burning a set of fuses. They are one time programmable i.e. once the devices are configured they cannot be altered. Quicklogic is the industrial manufacturer for anti-fuse based FPGA devices. The FPGA devices using the flash programming technology may re-programm the device several thousand times. The devices using flash programming technology are non-volatile i.e. they keep their configuration after the power-off. This technology is expensive and takes several seconds for reconfiguration. The flash based devices are manufactured by the Actel corporation.

The current dominating programming technology for FPGAs is SRAM based technology. It features fast configuration and unlimited times re-programming of the digital circuit. The devices using SRAM programming technology are volatile and may require additional circuitry to load configuration in the device after power is on. Due to the lower system cost, unlimited and fast re-programmable features of programming a device, the FPGA device using SRAM based programming technology is used in this research.

The main vendors for making SRAM based FPGA devices are Xilinx and Altera. The Xilinx, Inc. is an American technology company. It is known for inventing the FPGAs. The Xilinx FPGA product families includes a variety of high performance (Virtex family), mid range (Kintex family) and low cost (Artix / Spartan family) devices. The Altera corporation is also an American manufacturer of FPGA devices. It produces companies high bandwidth devices (stratix series) with 1.1 million logic elements. The Altera corporation also produces low cost and low power FPGA devices known as cyclone series and SoC FPGAs. The Arria series FPGA devices are produced by Altera for balance in power, cost and performance of FPGA devices. The difference between the Altera and Xilinx FPGA devices is of the internal structure. The Xilinx FPGA device uses CLB (complex logic blocks) and Altera FPGA devices uses logic cells. The CLBs are built with LUTs, flip flops and multiplexers whereas the logic cells are made up by multiplexers alone.

There are number of families of FPGA available, offering different levels of complexity in size and logic resources. The research in this thesis requires a lower cost SRAM based FPGA device to re-programm / alter a digital circuit unlimited times. Xilinx Spartan and Altera Cyclone families are both excellent devices featuring the lower cost families of SRAM based FPGA devices. The research in this thesis does not favour any specific family for implementing a digital circuit on FPGA devices. However, due to ease in comparing implementation results with other recent and old published papers, a Xilinx Spartan3 and Spartan6 families are being chosen for implementation of logarithmic converters and logarithmic based correlation applications in this research.

### 2.4.1 Altera Cyclone Family Overview

The Cyclone SRAM based FPGA family is based on a $1.5V$, $0.13\mu m$, all-layer copper SRAM process, with densities up to 20,060 logic elements (LEs) and up to 288 kbits of RAM. With features like phaselocked loops (PLLs) for clocking and a dedicated double data rate (DDR) interface to meet DDR SDRAM and fast cycle RAM (FCRAM) memory requirements, Cyclone devices are a cost-effective solution for data-path applications. Cyclone devices support various I/O standards, including LVDS at data rates up to 311 megabits per second (Mbps) and 66-MHz, 32-bit peripheral component interconnect (PCI), for interfacing with and supporting ASIC devices. Altera also offers new low-cost serial configuration devices to configure Cyclone devices. A detailed overview of Cyclone family compared to Logic elements, Block RAM, PLLs and I/O pins are shown in table 2.1.

Table 2.1: Cyclone FPGA Family Overview [20]

| Device | Logic Elements | Block RAM | Total RAM bits | PLLs | I/O pins |
|---|---|---|---|---|---|
| EP1C3 | 2,910 | 13 | 59,904 | 1 | 104 |
| EP1C4 | 4,000 | 17 | 78,336 | 2 | 301 |
| EP1C6 | 5,980 | 20 | 92,160 | 2 | 185 |
| EP1C12 | 12,060 | 52 | 239,616 | 2 | 249 |
| EP1C20 | 20,060 | 64 | 294,912 | 2 | 301 |

### 2.4.2 Xilinx Spartan3 Family Overview

The Spartan3 family of FPGA devices was made as a successor of the SpartanIIE family. Spartan3 devices are being widely used for a range of consumer electronic applications because of their low cost on a comparison to the other families of FPGA. The members of this family provide system gates from 50k to 5M (million) in number. The Spartan3 family CLBs are made of RAM based 4 input LUT, shown in figure 2.2. For applications, multipliers used can either be made by using LUT available on the slices of Spartan3 device or by using embedded 18 bit dedicated multipliers. The Spartan3 family provides Block RAM for data storage in the size of 18k bits dual port blocks [21]. A detailed overview of Spartan3 family compared to Logic cells, CLBs, Distribute RAM, Block RAM and Dedicated multipliers are shown in table 2.2.

Table 2.2: Spartan3 FPGA Family Overview [21]

| Device | System Gates | Logic Cells | Total CLBs | Distributed RAM | Block RAM | Dedicated Multipliers |
|---|---|---|---|---|---|---|
| XC3S50 | 50k | 1,728 | 192 | 12k | 72k | 4 |
| XC3S200 | 200k | 4,320 | 480 | 30k | 216k | 12 |
| XC3S400 | 400k | 8,064 | 896 | 56k | 288k | 16 |
| XC3S1000 | 1M | 17,280 | 1,920 | 120k | 432k | 24 |
| XC3S1500 | 1.5M | 29,952 | 3,328 | 208k | 576k | 32 |
| XC3S2000 | 2M | 46,080 | 5,120 | 320k | 720k | 40 |
| XC3S4000 | 4M | 62,208 | 6,912 | 432k | 1,728k | 96 |
| XC3S5000 | 5M | 74,880 | 8,320 | 520k | 1,872k | 104 |

For the purposes of testing in this thesis a Spartan3 Starter Kit board from Digilent Inc. is used. The Starter Kit board [22] provides a 20,000 gate Xilinx Spartan3 XC3S200 device with four digit seven segment LED display, 3-bit 8-colour VGA display port, ps2 port, 9 pin rs232 serial port with slide and push switches (shown in figure 2.4 ).

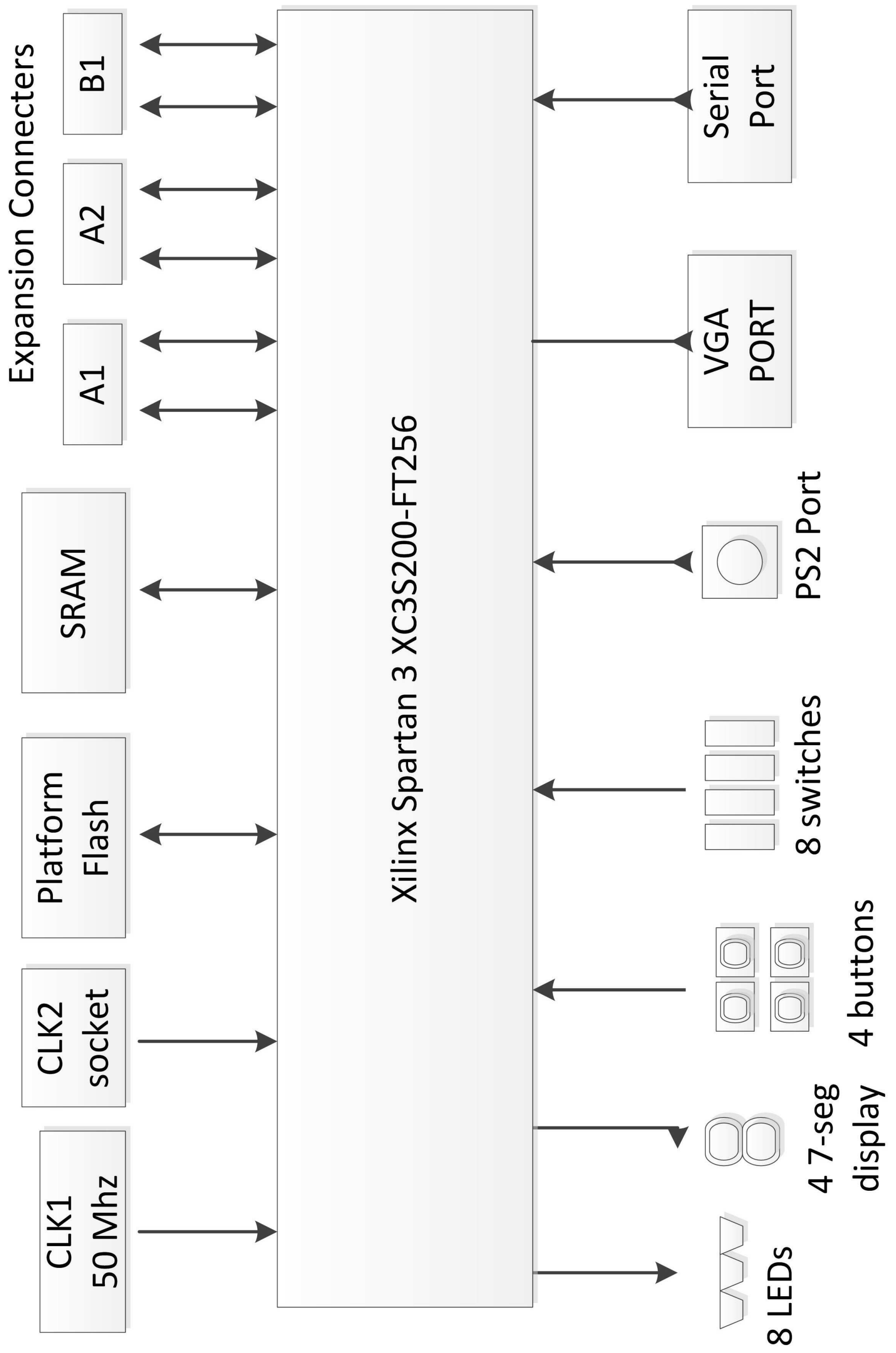


Figure 2.4: Spartan 3 Starter Kit Board [22].

### 2.4.3 Xilinx Spartan6 Family Overview

The Spartan6 family consists of 13 members ranging from 3k to 147k logic cells. The Spartan6 device CLBs use dual register 6 input LUT architecture (shown in figure 2.3). The Block RAM available on the device is further reduced to 9k bits blocks in comparison to previous spartan families. Instead of using dedicated 18 bit multipliers, as in the case of previous Spartan3 families, the Spartan6 device uses DSP48A1 slices. DSP48A1 blocks perform operations like multiply, add, multiply add and multiply accumulate, which are essential for DSP applications. The DSP48A1 slice in their architecture consists of two input pre-adder/subtractor, 18 bit two's complement multiplier with a full precision of 36 bit result and two input 48 bit post-adder/subtractor. Figure 2.5 shows a detailed diagram of the DSP48A1 slice [23].

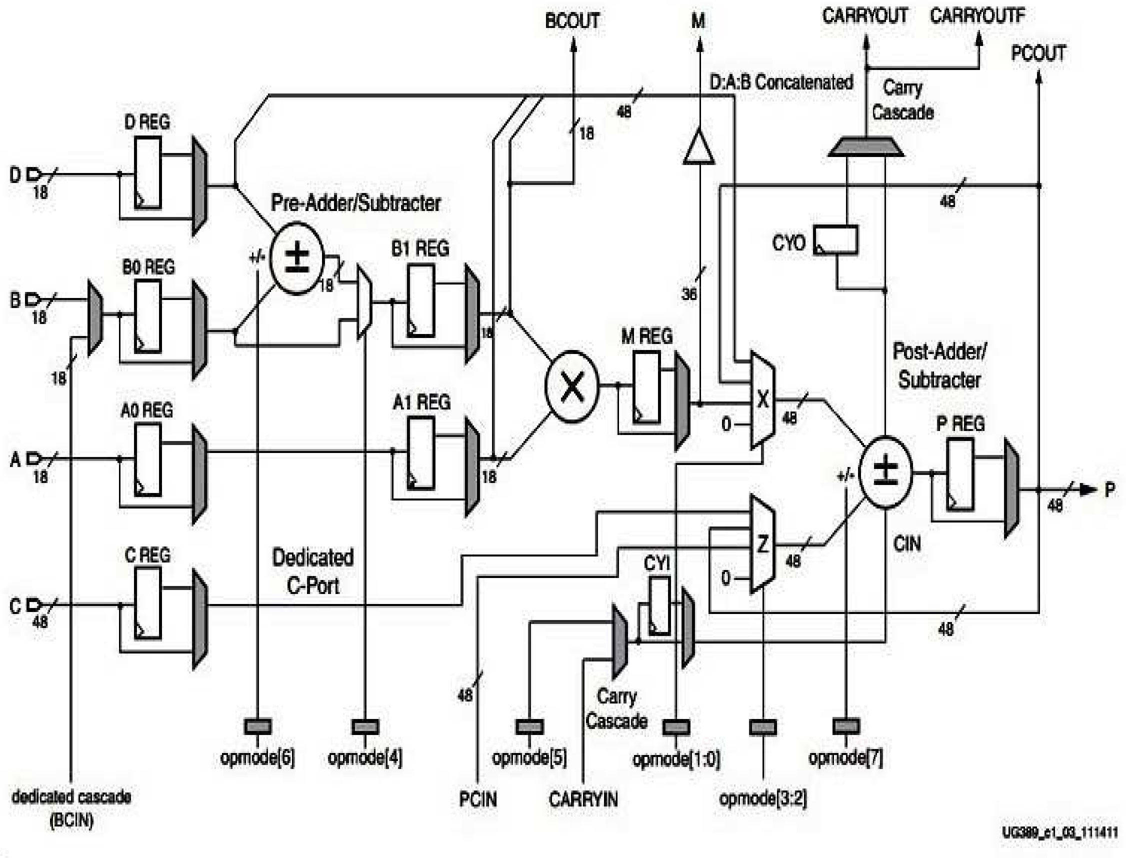


Figure 2.5: DSP48A1 Slice [23].

A detailed overview of the Spartan6 family with a comparison of their Logic cells, CLBs, Distribute RAM, Block RAM and DSP48A1 slices is shown in table 2.3 [24].

Table 2.3: Spartan6 FPGA Family Overview [24]

| Device | Logic Cells | CLBs Slices | Distributed RAM | Block RAM (18 kb) | DSP48A1 Slices |
|---|---|---|---|---|---|
| XC6SLX4 | 3,840 | 600 | 75k | 12 | 8 |
| XC6SLX9 | 9,152 | 1,430 | 90k | 32 | 16 |
| XC6SLX16 | 14,579 | 2,278 | 136k | 32 | 32 |
| XC6SLX25 | 24,051 | 3,758 | 229k | 52 | 38 |
| XC6SLX45 | 43,661 | 6,822 | 401k | 116 | 58 |
| XC6SLX75 | 74,637 | 11,662 | 692k | 172 | 132 |
| XC6SLX100 | 101,261 | 15,822 | 976k | 268 | 180 |
| XC6SLX150 | 147,443 | 23,038 | 1,355k | 268 | 180 |
| XC6SLX25T | 24,051 | 3,758 | 229k | 52 | 38 |
| XC6SLX45T | 43,661 | 6,822 | 401k | 116 | 58 |
| XC6SLX75T | 74,637 | 11,622 | 692k | 172 | 132 |
| XC6SLX100T | 101,261 | 15,822 | 976k | 268 | 180 |
| XC6SLX150T | 147,443 | 23,038 | 1,355k | 268 | 180 |

For the testing purposes in this thesis a Nexys3 board from Digilent Inc. is used. The Nexys3 board provides Xilinx Spartan6 XC6SLX16 device with Adept USB2, cellular RAM of 16 Mbyte, parallel PCM nonvolatile of 16Mbyte memory, ethernet port, 8 bit VGA, USB HID Host and UART port, basic input output LEDs, push and switch buttons (shown in figure 2.6).

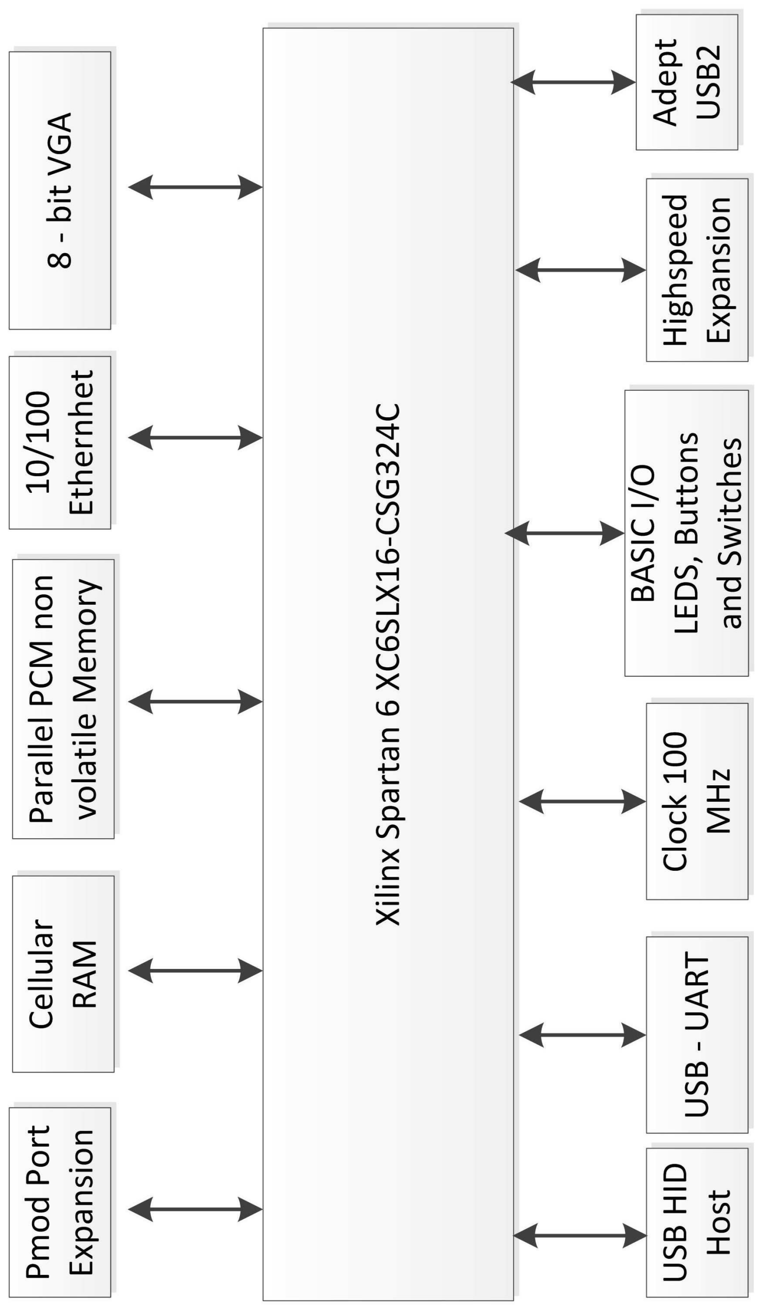


Figure 2.6: Nexys3 Board [25].

## 2.5 Hardware Description Language

In computer languages, hardware description language is used to program an FPGA device. It describes the behaviour and structural flow of digital logic circuits. The two common form of hardware description languages used in industries are Verilog and VHDL. For the research in this thesis VHDL is used for implementing novel logarithmic converters on FPGAs.

The VHDL stands for very high speed integrated circuit hardware description language. It was first introduced in 1981 by the U.S. department of defence (DoD). The VHDL is used to write and synthesis digital logical circuits. The simulation programs such as Xilinx Vivado and Mentor Graphics ModelSim are used to test the logic design written in VHDL codes. After the successful simulation of digital logic circuits, the circuits are implemented on the hardware FPGA devices by translating synthesis output into a bitstream suited for a specific target device.

The advantage of using VHDL for system design is that it verifies the system through simulation before synthesis tools implements the circuit on hardware. Another benefit of using VHDL over other conventional C, BASIC and assembly languages is that it provides a description of concurrent process rather than sequential process.

## 2.6 Numerical Data Representation on FPGAs

The mathematical operations like addition, subtraction, multiplication and division in hardware uses either fixed point binary number system or floating point binary number system. The fixed point number system is straightforward and does not create any complexity in mathematical operations as numbers of integers and fraction bits are pre-defined.

The floating point on the other hand uses exponent bits and fraction bits covering a large range of numbers with limited precision in comparison to the fixed point number system. There is a trade off between the floating point number system and fixed point number system on the basis of precision achieved and range covered.

The IEEE-754 [9] standard defines binary floating point numbers into single and double precision. The single precision uses 32 bits out of which 1 bit is for sign bit, 8 bits for exponent and 23 bits for fractional part (shown in figure 2.7). The double precision (64 bits) in a floating point number are divided as 1 bit for sign bit, 11 bits for exponent and 52 bits for fractional part (shown in figure 2.8).

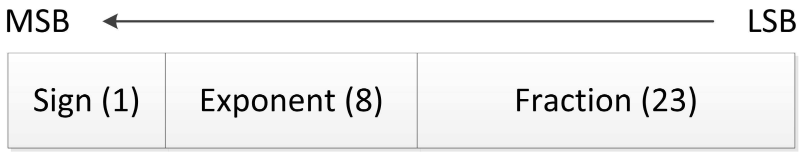


Figure 2.7: IEEE-754 std Single Precision.

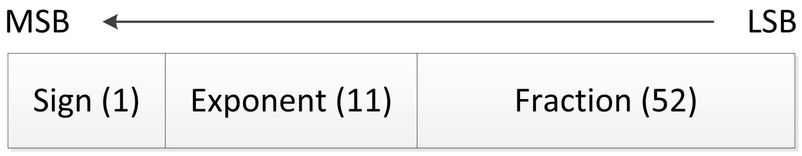


Figure 2.8: IEEE-754 std Double Precision.

The DSP applications are mostly real time based, requiring a large dynamic range. Using the floating point number system on an FPGA hardware has been achieved in [26–29]. However, multiplication and division operations of a floating point number on FPGAs is not trivial as described in [30] and [31]. Multiplication of floating point numbers further requires operations of multiplication of fractional bits and addition of exponent bits. Similarly the division of floating point numbers requires the division of fractional bits and subtraction of exponent bits, making computation slower in comparison to the fixed point number system.

A floating point adder/subtractor algorithm [30] is presented on an FPGA device in order to maximize speed and minimise area. In the algorithm [30] addition and subtraction are performed in three stages. A floating point number is represented as

$$v = (-1)^S.2^E.(1.F) \tag{2.1}$$

where $S$, $E$ and $F$ are used to represent sign, exponent and fraction/mantissa fields of the floating point number. Figure 2.9 shows a three-stage 18-bit adder, where addition takes place between $v_1$ and $v_2$ floating point numbers. In the first stage, alignment of fraction/mantissa bits is performed by shifting to the right. The number of positions to shift the fraction bit right is decided by subtracting exponent bits. The subtraction of exponent bits is decided by a comparison of absolute values of $v_1$ and $v_2$ floating point numbers. The addition/subtraction of fraction bits takes place in the second stage, depending on the sign bit. In stage three, normalisation of the resultant fraction bit is performed, by shifting fraction bits to the left until high order bit is one.

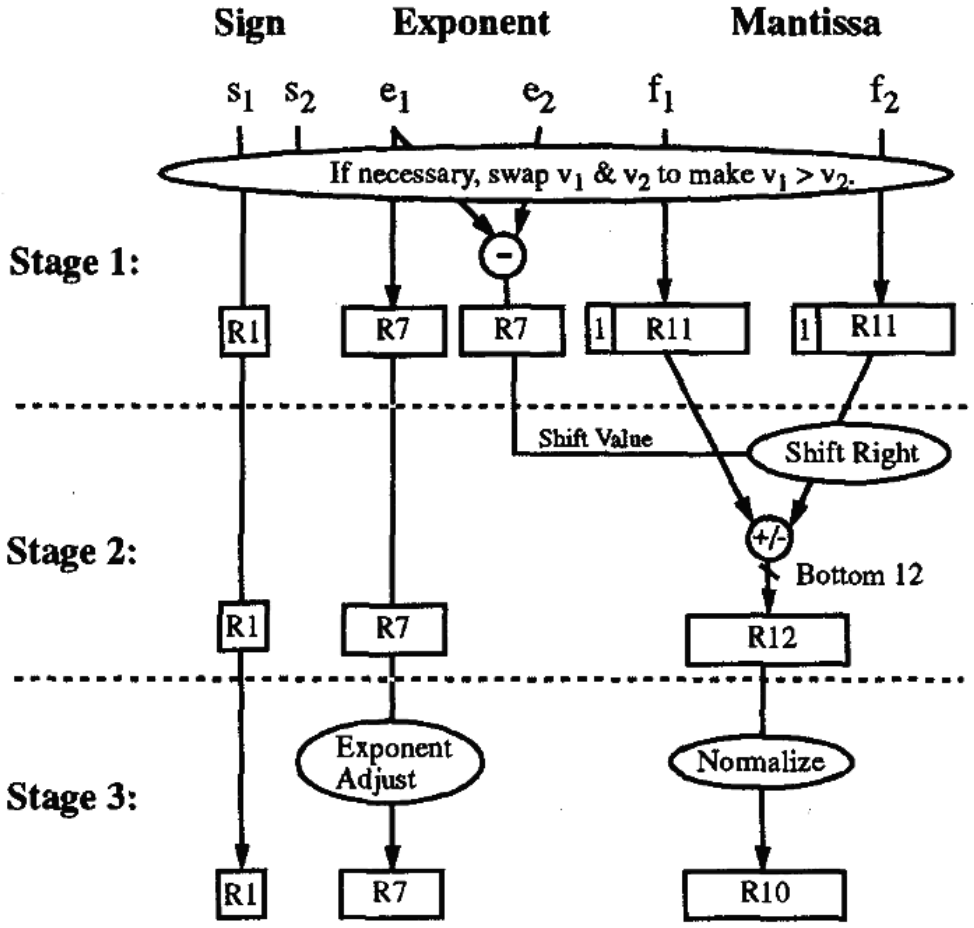


Figure 2.9: Three stage 18-bit floating point adder [30].

[30] also presents a similar three-stage 18-bit multiplier unit. In stage 1, the addition of exponent bits of floating numbers take place and '1' is concatenated to the left side of fraction bits. The multiplication of two fractional bits is performed in stage 2 by using an integer multiplier. The exponent part of the floating point

number is adjusted depending on the higher bit of the multiplication result. Resultant sign bit is calculated by using a xor gate. Normalisation of the multiplication resultant is performed in stage 3. A three-stage 18-bit floating point multiplier is shown in figure 2.10. The integer multiplier used for fraction bits in multiplication unit suffers the problem of bottleneck. The author uses four different methods to optimise the integer multiplier by using a integer multiplier available in VHDL compiler, array multiplier and by using pipeline in multiplication design to increase the speed of the design.

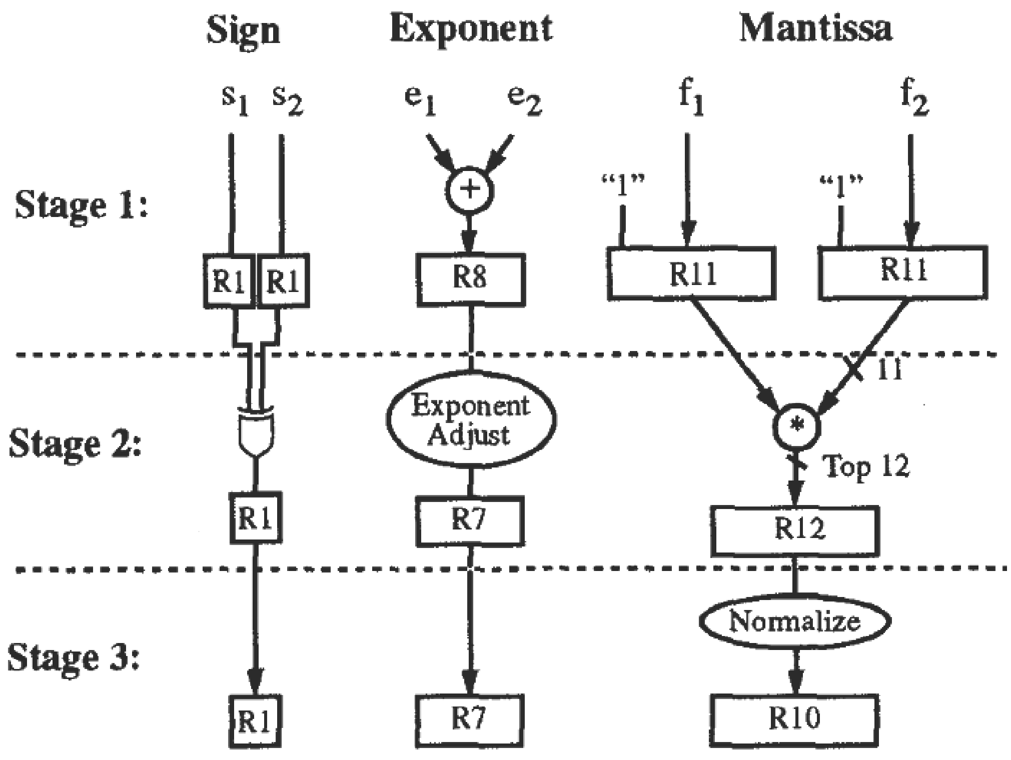


Figure 2.10: Three stage 18-bit floating point multiplier [30].

In [27], the author describes an addition algorithm of floating point numbers, shown in figure 2.11. The hardware implementation of addition algorithm is shown in figure 2.12. The Initial version of 32-bit adder design took 72% of area on an Altera 81188 device. The author claimed reduction of area by 25% when FLEX 8000 logic elements were used in the circuit. For the multiplication of floating point numbers , the author [27] uses a digit serial multiplier. A digit serial multiplier gives performance in between a bit serial multiplier [32] and a bit parallel multiplier [33]. A digit serial multiplier is implemented on an Altera FLEX 81188 device. Figure 2.13 shows arrays of bit multipliers (BM). In the multiplication, multiplier bits are passed to columns of array and multiplicand bits are passed to rows. The multiplication unit consumes 49% of the Altera flex 81188 device, giving a performance of 2.3 MFlops.

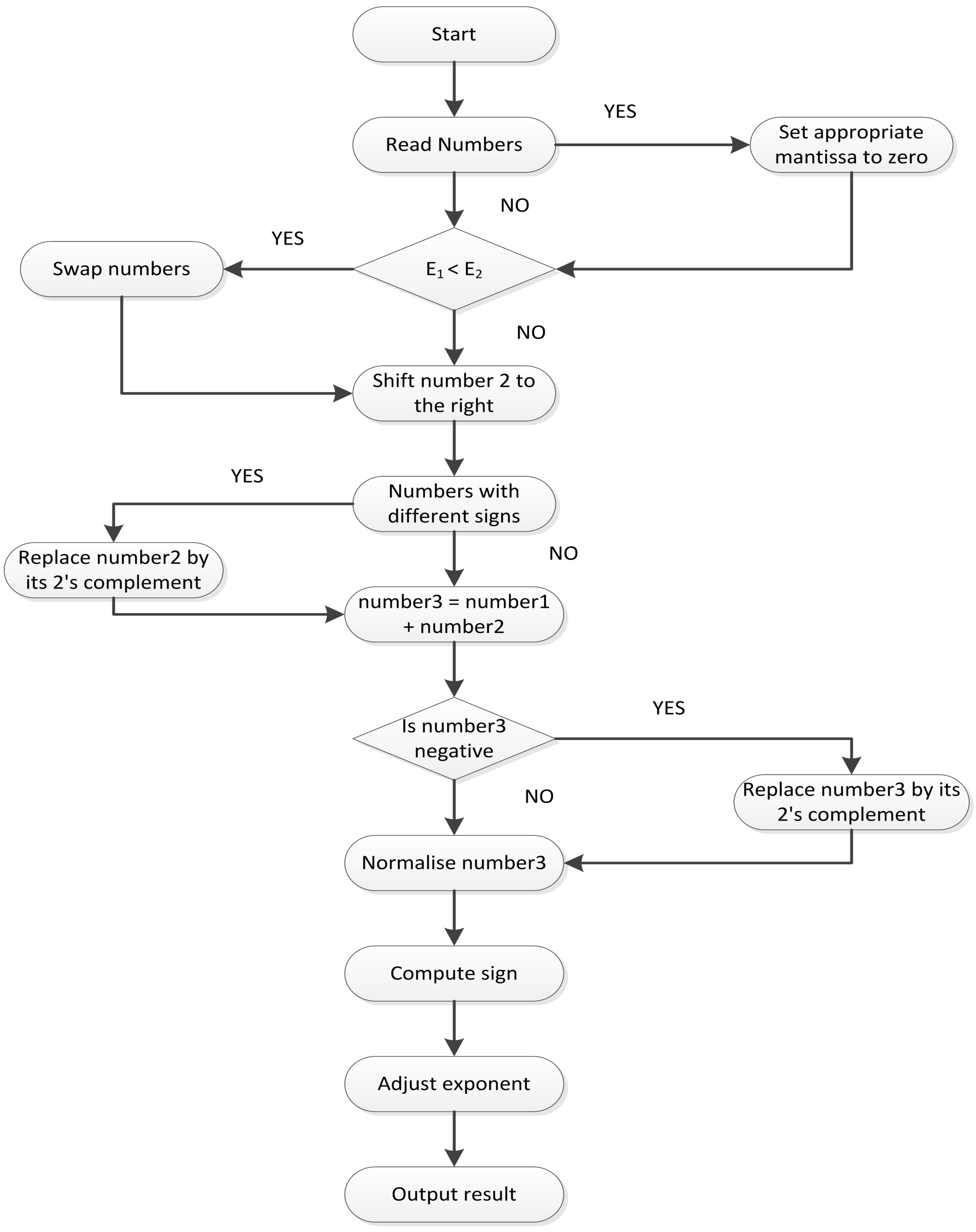

Start
YES
Read Numbers
Set appropriate mantissa to zero
NO
YES
Swap numbers
E1 < E2
NO
Shift number 2 to the right
YES
Numbers with different signs
Replace number2 by its 2's complement
NO
number3 = number1 + number2
YES
Is number3 negative
NO
Replace number3 by its 2's complement
Normalise number3
Compute sign
Adjust exponent
Output result

Figure 2.11: Floating point addition algorithm [27].

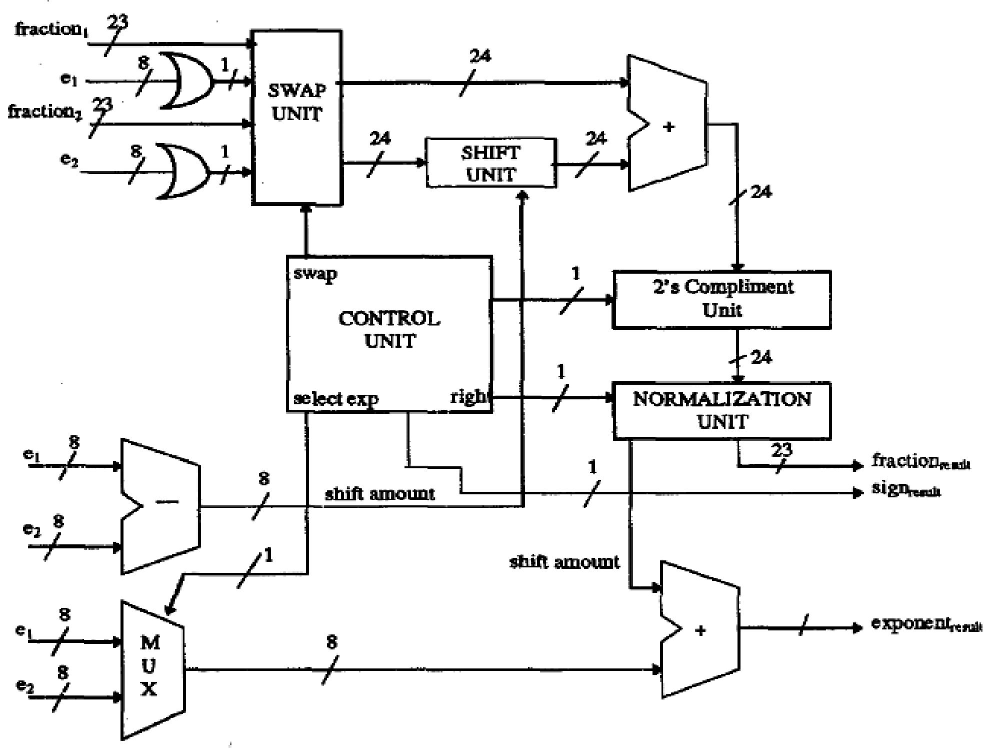


Figure 2.12: 32 bit floating point adder [27].

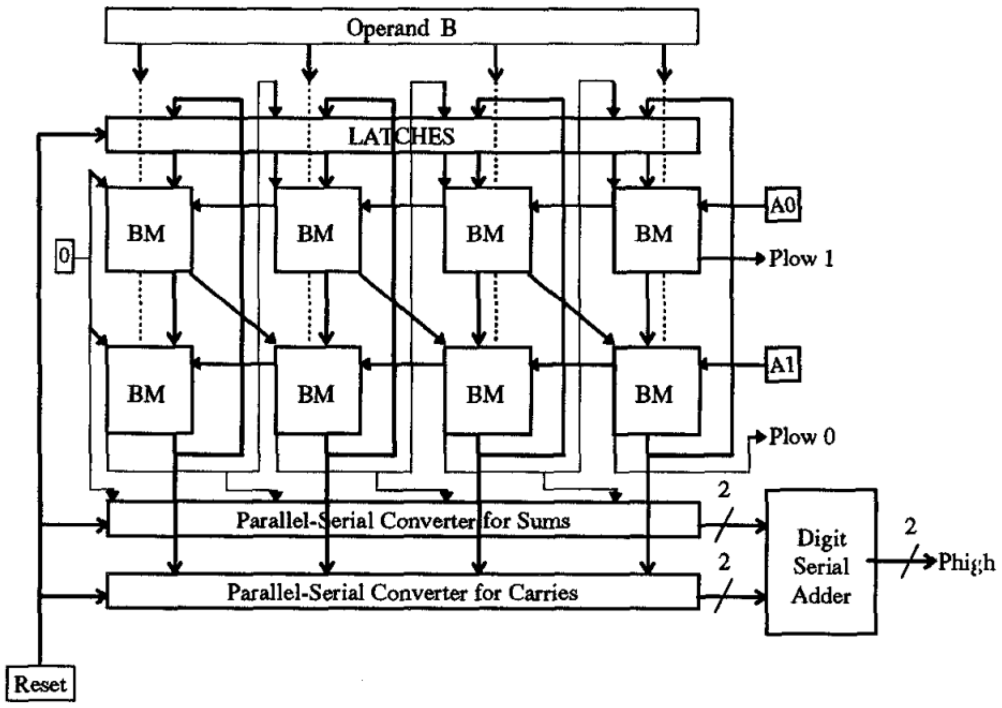

Operand B
LATCHES
BM
0
A0
Plow 1
A1
Plow 0
Parallel-Serial Converter for Sums
Parallel-Serial Converter for Carries
2
2
Digit Serial Adder
2
Phigh
Reset

Figure 2.13: Digit serial multiplier [27].

Another format of binary number system, LNS (logarithmic number system) not being widely used in hardware, solves the problem of multiplication and division in wide dynamic range [34]. LNS provides a similar range and precision to that of the floating point number system. In 2005, Haselman [31] compared floating point and logarithmic number systems on FPGAs. Tables 2.4 and 2.5 are reproduced from [31], showing a comparison of FPGA hardware (Virtex II 2000) resources used when arithmetic operations of multiplication and division are performed.

Table 2.4: Hardware usage in multiplication of single and double floating point precision [31]

| | Single FP Precision | | Double FP Precision | |
|---|---|---|---|---|
| | FP | LNS | FP | LNS |
| Slices | 297 | 20 | 820 | 36 |
| Multipliers | 4 | 0 | 9 | 0 |
| 18k BRAM | 0 | 0 | 0 | 0 |
| Latency(ns) | 65 | 10 | 83 | 12 |

Table 2.5: Hardware usage in division of single and double floating point precision [31]

| | Single FP Precision | | Double FP Precision | |
|---|---|---|---|---|
| | FP | LNS | FP | LNS |
| Slices | 910 | 20 | 3376 | 36 |
| Multipliers | 0 | 0 | 0 | 0 |
| 18k BRAM | 0 | 0 | 0 | 0 |
| Latency(ns) | 150 | 10 | 350 | 12.7 |

## 2.7 Summary

This chapter has summarised the FPGA technology for designing a digital logic circuit. The FPGAs can implement simple and complex function on a single device. The commonly used functions (adder, multiplier etc.) on modern FPGAs are present in dedicated hardware. The dedicated hardware resources can be combined with other existing logic design on FPGA board.

The number systems for arithemtic operations on FPGA devices are discussed and compared. The tables 2.4 and 2.5 shows that multiplication and division linear operations in LNS are very efficient in terms of hardware resources usage in FPGAs. However, before performing the linear operations of multiplication and division, a floating point number must be converted into a logarithmic number and vice versa with the resultant value. The conversion of floating point number to logarithmic number, while maintaining its accuracy the same as single floating point precision is further discussed in Chapters 3, 4 and 5. An example of using a logarithmic number system instead of floating point number in a real time based application is showed in Chapter 6.

# Chapter 3

# Logarithmic Conversion Literature Review

## 3.1 Introduction

This chapter provides an overview of algorithms and techniques used in converting linear to logarithm value with base 2 (Lin2Log) and vice versa (Log2Lin). Logarithmic conversion and logarithmic signal processing have been used in a number of applications and are of increasing interest because of the potential for area and higher resolution architectures [1, 35–37]. As a consequence, many algorithms and hardware architectures have been proposed over the past 50 years for converting a normalised binary number $x$ into a binary logarithm ($log_2 x$) or converting the binary exponent ($2^{log_2 x}$) back to a normalised binary number. These algorithms vary widely in terms of their accuracy, efficiency and speed. Over such a long time-span there have also been significant changes in the performance of the implementation technologies, resulting in newer architectures capable of achieving increasing conversion accuracies and making the logarithmic number system (LNS) a viable alternative in a number of modern applications [5, 6, 38–43].

As compared to recent algorithms early algorithms have low-precision, limited to 12-bits because of limited technology as integrated digital circuits their have limited memory and/or logic capacities. Recent publications show that algorithms which rely on direct LUT are more accurate, but conversion becomes impractical and large when using for 16-bits or higher because of their LUT size and access time [44]. At this time when higher precision is required curve fitting techniques such as Taylor, CORDIC, Chebyschev etc. are used to approximate the curve, using LUT for their coefficient values. There are algorithms based on curve fitting techniques using first order and second order approximation. Higher order polynomial approximation methods are used in few papers where the precision required is greater than IEEE single-precision floating point format [9].

The process of converting a fixed/floating point number into a logarithmic number is divided into two parts. The first part includes calculation of the integer part into a logarithmic characteristic value and the second part calculates the decimal/mantissa part into a logarithmic mantissa/fractional value. The finding of the characteristic/integer value of a logarithmic number from a normalised fixed/floating point number has been reasonably straightforward as there are similarities found in normalised floating point formats and logarithmic formats. In equations 3.1 and 3.2, $S$ denotes the sign bit. The exponent part of the floating point number ($E$) has the same value as the characteristic value or integer part ($I$) of the logarithmic value.

Floating Point Format

$$x = (-1)^S.2^E.(1.M) \tag{3.1}$$

Logarithmic Format

$$x = (-1)^S.2^I.2^F \tag{3.2}$$

So the initial step for finding a logarithmic number of a given fixed/floating point number is to normalise the input number. To normalise the input number by using a leading one or leading zero detector is discussed in detail in section 3.2. Once the input number is normalised and the characteristic value of the logarithmic number is obtained, the next step is to perform a Lin2Log conversion on the decimal part ($M$) of the floating/fixed point number. The different types of algorithms used for converting a normalised floating/fixed point number into a logarithmic number fraction/mantissa ($F$) are discussed in section 3.3.

## 3.2 Normalisation Methods

The first step in converting a fixed/floating point number to a logarithmic number is to normalise the input number. The normalisation is performed by using leading one detector (LOD) or leading zero detector (LZD) [45–52]. LOD/LZDs are preferred in designing a normalisation circuit as they are trivial to implement in hardware. LOD/LZDs work on simple algorithm of shifting the first one or non zero bit to the left-most bit position. A few popular and recent algorithms for leading one/zero detectors are explained in this section.

### 3.2.1 Leading Zero Detector

The leading zero detector circuit was described in a 1993 electronics letter [53], whose implementation was based on an algorithmic approach resulting in a modular scalable circuit for any number of bits. The LZD works by shifting bits to the left position until a nonzero bit is reached at MSB. The exponent part of the normalised number is calculated by decrementing the original exponent part by the total number of shifts until the first nonzero digit is reached. Figure 3.1 shows an Oklobdzija leading zero circuit which is in hierarchical order, consisting of valid and position

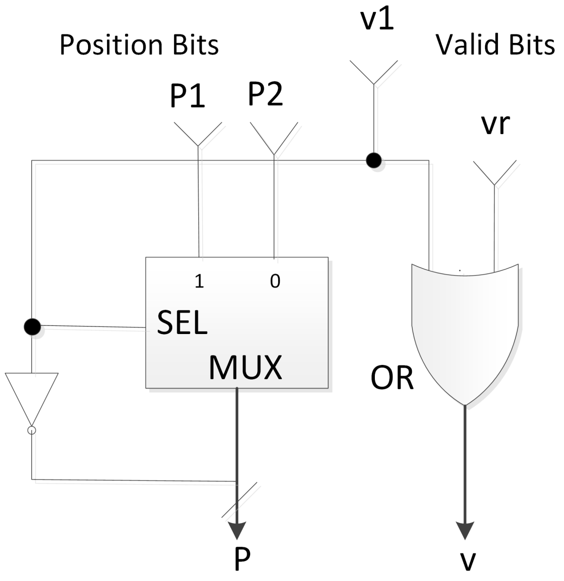


Figure 3.1: Oklobdzija leading zero detector circuit [53]

bits. When input is two bits it is trivial but with a 4-bit circuit, logic levels become two, and in the second level a valid bit is formed as the logical OR of the valid bits from the previous level. If all of the groups do not show a valid output, it means they are a string of zeros. This circuit is good for a small number of bits, as the number of bits increases output of this circuit becomes slower and more hardware is needed.

### 3.2.2 Leading One Detector

In an electronics letter [54] in 1998 a modular circuit for determining the leading one in a binary word is described. The circuit was designed for encoding binary data into a binary logarithm format, but it can also be used for floating point normalisation.

In figure 3.2 LOD is used. The output of the circuit is encoded using a small PLA (programmable logic array) to give the integer value of the logarithm and also to control a shifter circuit which is used to generate a fractional part of the exponent.

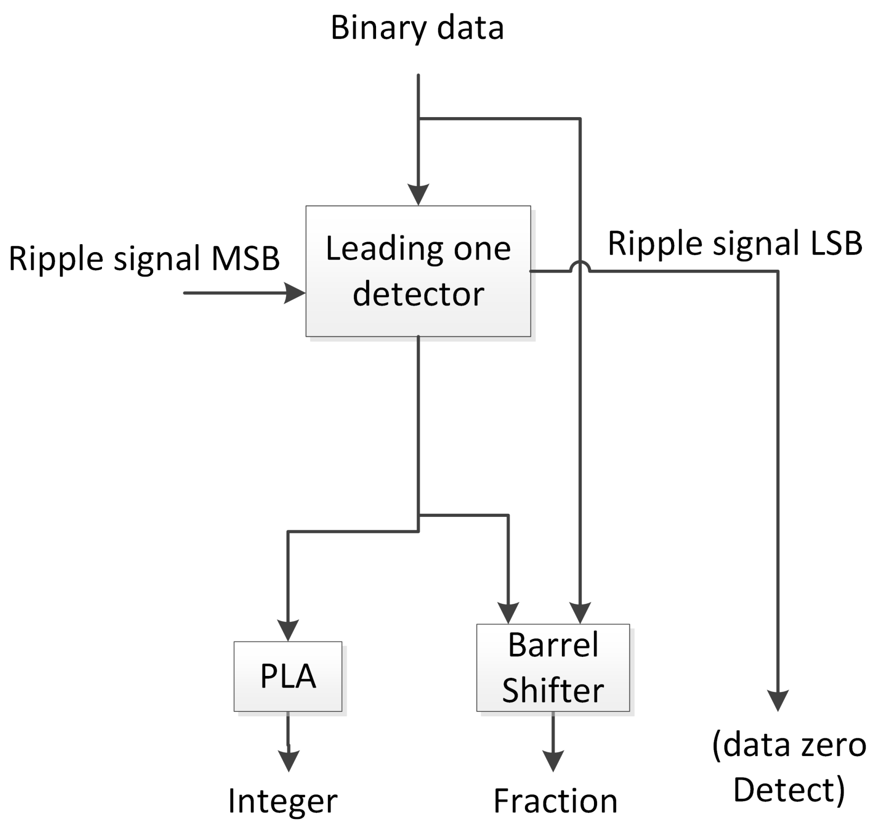


Figure 3.2: P. Lee's LOD circuit [54]

The approach for leading one detection in this letter contrasts with LZD described by Oklobdzija in 1993 [53] which uses a more complex circuit to produce the integer value directly, without the need for an encoding PLA. The circuit shown in figure 3.2 was designed and built using 1.2 $\mu m$ CMOS technology. The circuit works at a frequency of 30 MHz and occupies an area of 0.05 $mm^2$.

Abed et al. [55–58] proposed two approaches to LOD approximation by improving Mitchell's algorithm [59]. Each approach is used to obtain 0.6 $\mu m$ CMOS VLSI implementations of 16, 32 and 64-bit LOD circuits. The approach to design a fast LOD is based on dividing the input binary word into groups, which evaluate in parallel and independently of each other, but evaluation within each group is performed serially. Figure 3.3 shows a circuit of 4-bit fast LOD circuit.

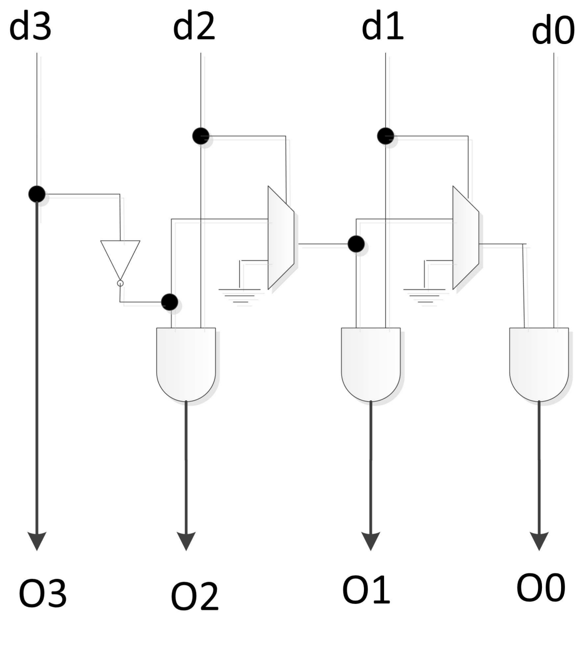


Figure 3.3: Fast 4-bit LOD [58]

For a higher bits LOD, the first stage consists of many 4-bit LODs, where each 4-bit LOD is used to evaluate each 4-bit of input data word. Therefore, the first stage requires $\frac{N}{4}$ 4-bit LODs (where $N$ denotes the total number of input bits) and because of the large number of LODs speed improves, but at a cost of more hardware and more power.

To eliminate this problem Abed et al. [58] proposed another type of approach which is made for low power and hardware efficiency. In this approach Abed et al. [58] used a single 4-bit LOD circuit to perform the operation of the first stage, instead of $\frac{N}{4}$ 4-bit LODs. In figure 3.4 a 16-bit LOD circuit is drawn for low power. In this 16 bits are divided into 4 groups of 4 bits, OR gates are used to determine whether a group has at least 1 or not which is then forwarded to the LOD to produce a 4-bit control word which determines the leading group. A 4x4 multiplexer is used which is controlled by control bits, each control bit controls a 4-bit MUXs row. The 16-bit LOD circuit generates 16-bits active-high decoded binary word that has leading one.

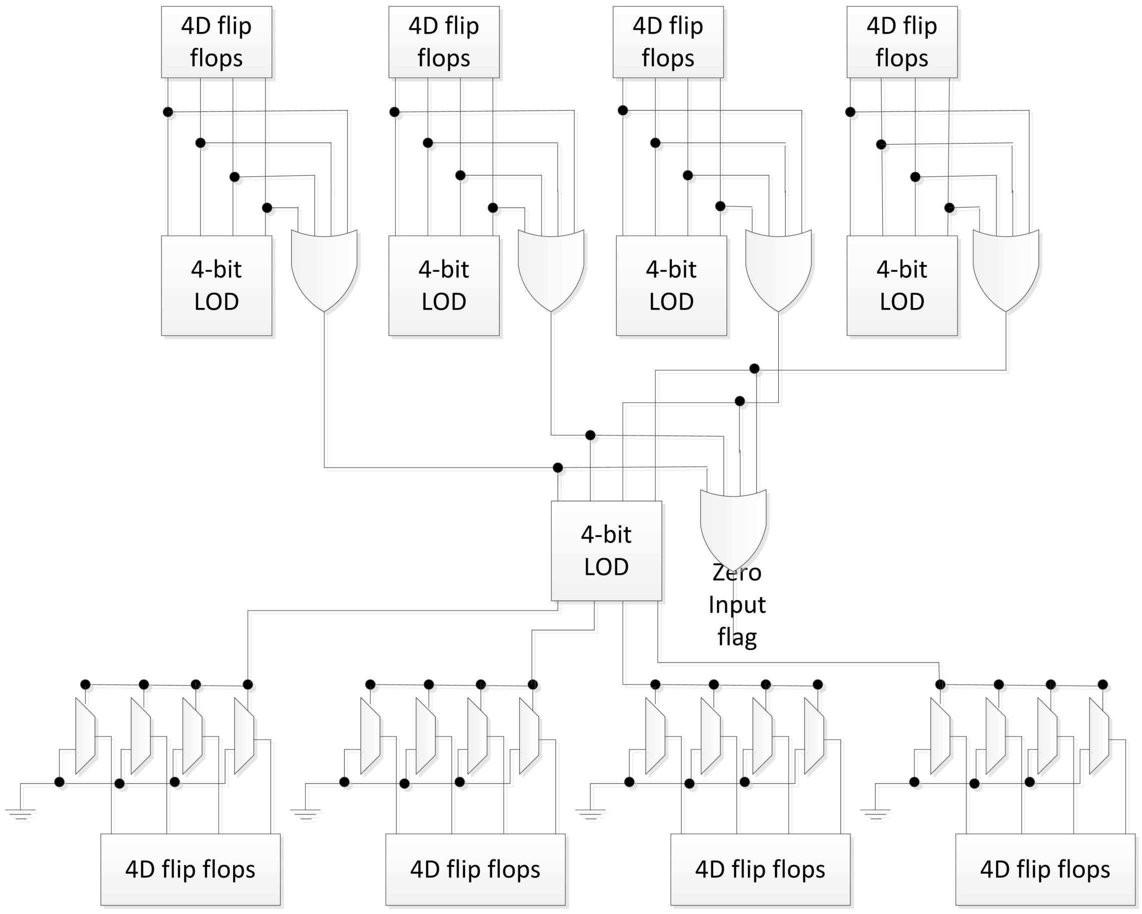


Figure 3.4: Fast 16-bit LOD [58]

## 3.3 Logarithmic Converter

Logarithmic conversion of the decimal part of normalised fixed/floating input numbers using different algorithms is mentioned in this section. Algorithms mentioned in this section discuss Lin2Log and Log2Lin conversions on the basis of bits precision achieved by them. In all the cases input numbers are assumed to be positive and for negative numbers they are dealt with by sign ($S$) magnitude arithmetic (see equations 3.1 and 3.2).

An analysis and comparison of different algorithms mentioned in this section is performed by MATLAB simulations. In simulations a normalised input number ($x$) is assumed. Here we consider the conversion of the normalised number $1 \leq 1.F < 2$, where $F$ is the fractional component of a normalised number (see equation 3.2). This chapter shows the accuracy achieved by Lin2Log and Log2Lin algorithms and discusses their error distribution.

### 3.3.1 Direct PWL Conversion Algorithms

This section of algorithms uses simple PWL approximation methods without using many hardware resources such as LUT etc. to calculate the logarithmic approximation of fixed/floating point input numbers. These algorithms achieved a limited amount of accuracy and are considered as the benchmark for new algorithms. One of the earliest and most frequently compared algorithms was proposed by Mitchell [59] in 1962. This paper is seen as the start of Lin2Log and Log2Liin processors. A simple addition or subtraction and shifting operation is all that is required to multiply or divide in this algorithm.

The approximations to binary logarithms are easy to generate using Mitchell's approximation [59]. No look-up tables are required, and multiplication and division operations are reduced to addition and subtraction operations. The algorithm proposed in Mitchell's approximation [59] uses a straight line interpolation between the points where mantissa is zero. The multiplication error in Mitchell's approximation [59] computation is -11.1% and division error is 12.5%.

To describe this algorithm an example is shown in figure 3.5. The algorithm is divided into four parts, considering A and B (8 bits each) as input bits. The first step in this algorithm is to shift input bits (A and B) to the left, until their most significant bit (having value one) is in the left-most position. Once the shifting is complete, the counter registers ($X_3X_2X_1$ and $Y_3Y_2Y_1$) contain the characteristics of the logarithm of A and B (this method also known as leading one detection).

Input registers A and B (6-0 bits) are shifted into bit positions (6-0 bits) of C and D registers. After shifting the C and D registers, they contain the logarithm of the original number. After obtaining the logarithmic value of the input registers, multiplication and division operations which are reduced to addition/subtraction in logarithmic domain take place. Registers C and D are considered as inputs for this operation and results are stored in register E. Register F contains the resultant approximated value (in linear domain) of two linear values (A and B). The resultant exponent part in linear domain is calculated by decoding bit values $Z_4Z_3Z_2Z_1$ in register E and placing one in appropriate position in F. The approximated fractional part of the resultant is calculated by shifting the remaining bits of register E (that is, 0-6 bits) into register F.

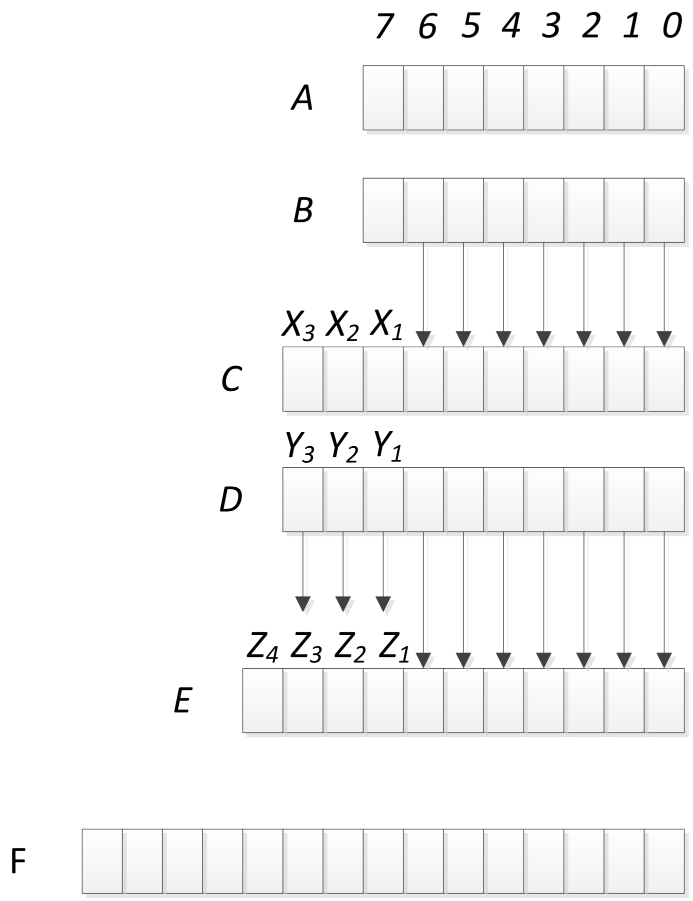


Figure 3.5: Mitchell's algorithm generating binary logarithm [59]

Mitchell's algorithm can be represented mathematically. Let a number in the linear number system (LNS) be represented as

$$N = 2^k(1 + x) \tag{3.3}$$

where $2^k$ $(k = 0, \pm 1, \pm 2...)$ is the exponent part of the number and $x$ be the fractional part of the number. The original logarithmic value of this number can be represented as

$$y = log_2(N) = k + log_2(1 + x). \tag{3.4}$$

The approximated value attained after using Mitchell's algorithm is

$$y' = log_2(N)' = k + x. \tag{3.5}$$

Conversion error, which is the difference between the original logarithmic value and the approximated value from the algorithm, is represented as $e$ (error).

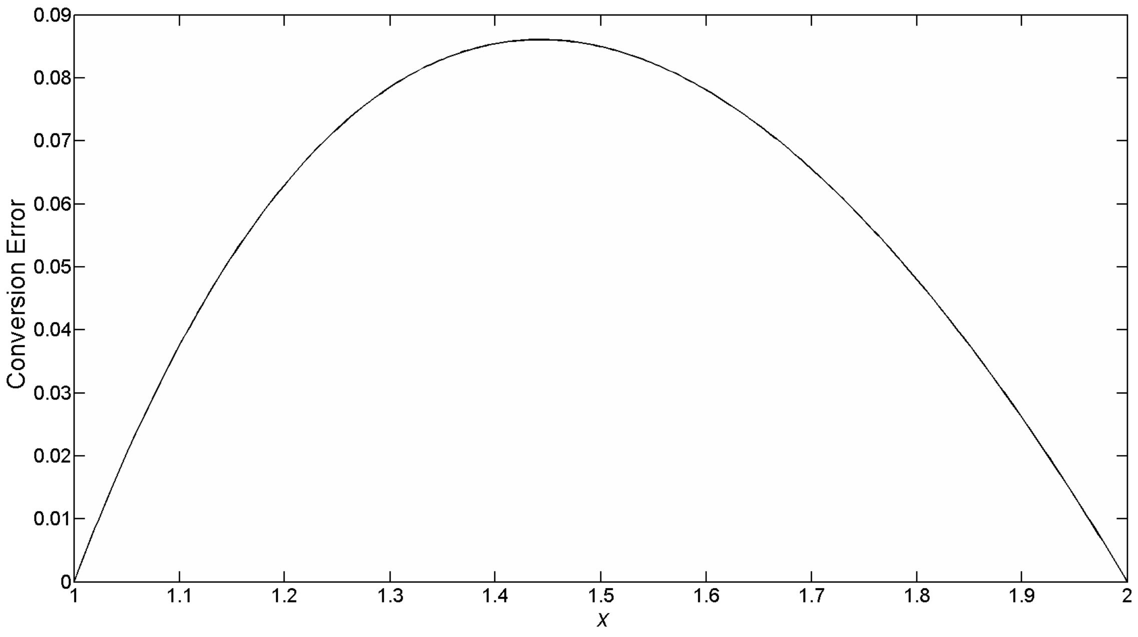
$$e = y - y'. \tag{3.6}$$

$$e = k + log_2(1 + x) - (k + x) = log_2(1 + x) - x. \tag{3.7}$$

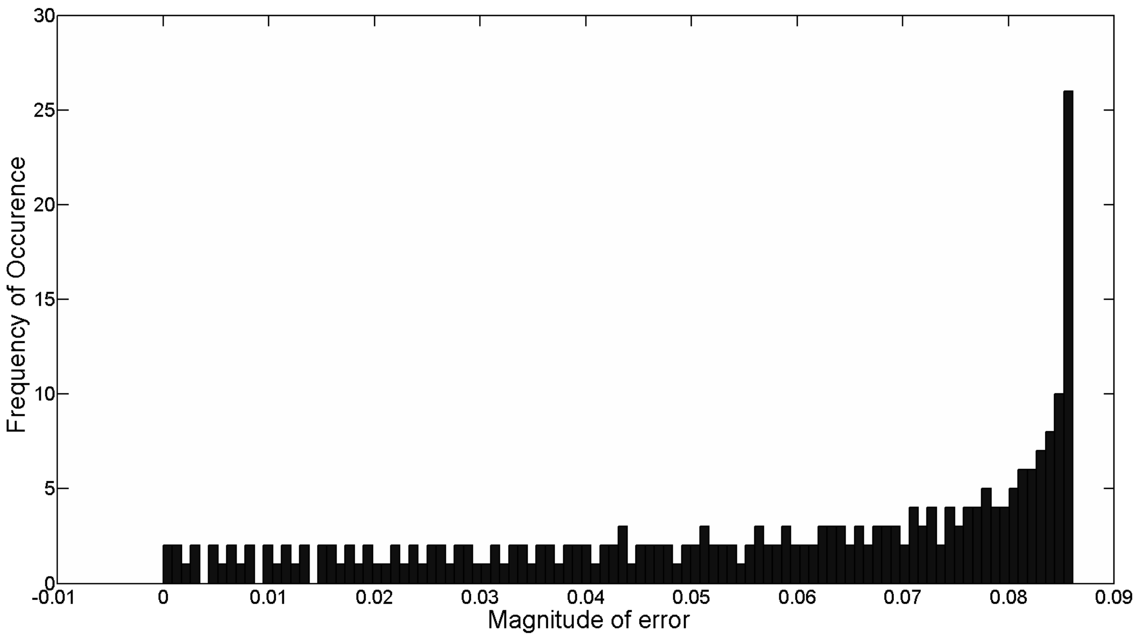
Error, $e$, range calculated after derivative equal to zero is

$$0 \leq e \leq 0.08639. \tag{3.8}$$

Hence the algorithm is equivalent roughly to four bits precision. Mitchell's approximation [59] conversion errors are simulated in MATLAB and are shown in figure 3.6.



(a)



(b)

Figure 3.6: Mitchell approximation [59] (a) Absolute error (b) Error histogram

Combet et al. [60] proposed an algorithm in 1965, which claimed to reduce the error of Mitchell's approximation [59] by a factor of six. The realisation involves not only counting and shifting but also binary decision-making and addition. The algorithm uses a piecewise linear approximation on the error curve produced by

Mitchell's approximation [59] of four equal segments. The mantissa is divided into four equal parts and these segments are using coefficients which are found by a trial and error method.

The Maximum error in Mitchell's approximation [59] is 0.086 (from equation 3.7) and the error range generated by the Combet et al. approximation [60] is 0.014 where the maximum and minimum error values are + 0.008 and - 0.006 (see figure 3.7). This algorithm was realised at that time as a basic part of a digital period-meter for a nuclear reactor.

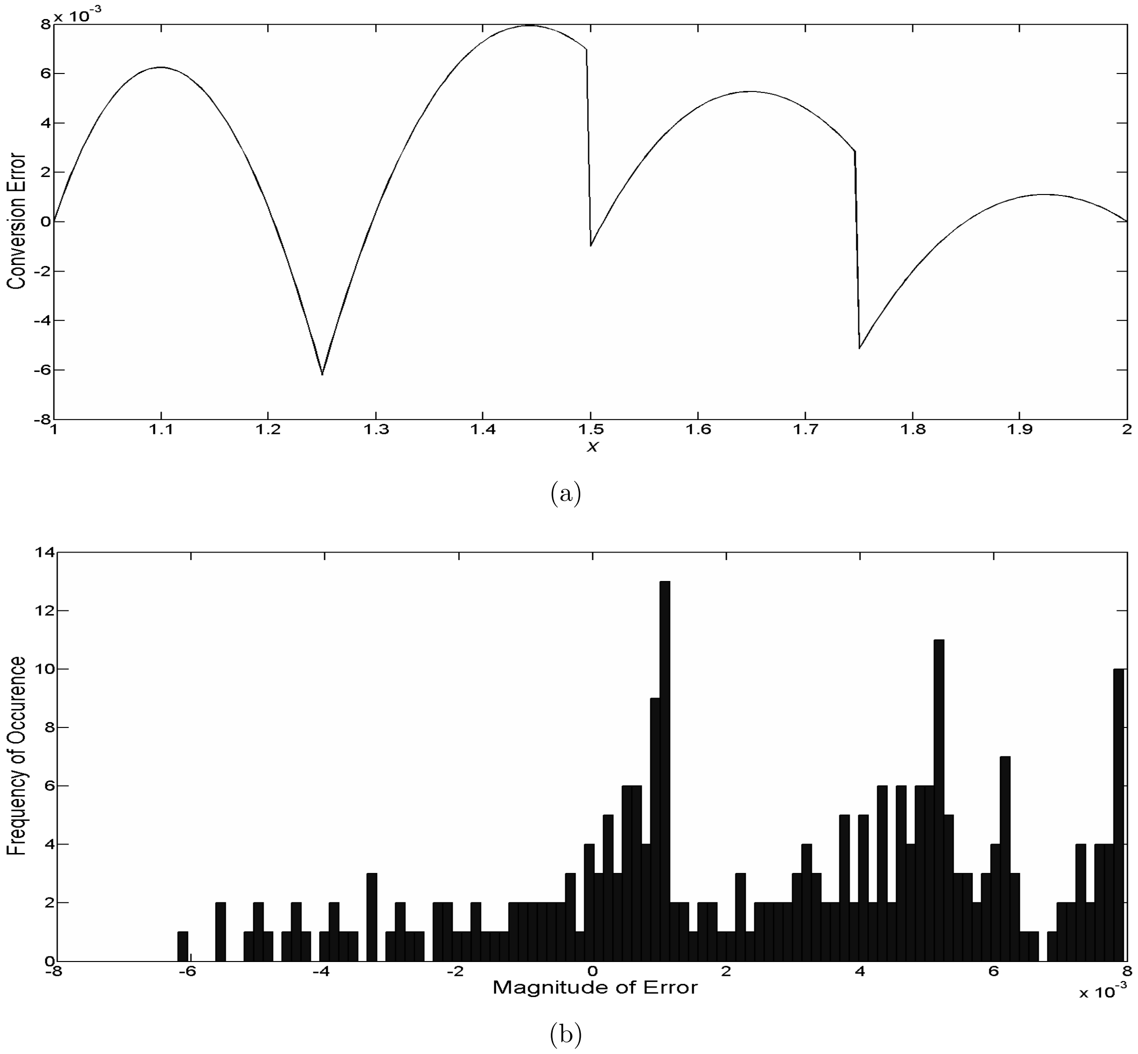


Figure 3.7: The Combet et al. Logarithmic approximation [60] (a) Absolute error (b) Error histogram

The method for reducing error consists of a piecewise linear approximation of

$log_2A(1+x)$ which is an approximated value of $log_2(1+x)$. The error generated by the Combet et al. approximation [60] can be written as

$$error = log_2(1+x) - log_2A(1+x). \tag{3.9}$$

where $log_2A(1+x)$ approximation in four regions is defined as

$$log_2A(1+x) = x + \frac{5}{16}x \qquad for\ 0 \le x \le \frac{1}{4}, \tag{3.10}$$

$$log_2A(1+x) = x + \frac{5}{64}x \qquad for\ \frac{1}{4} \le x \le \frac{1}{2}, \tag{3.11}$$

$$log_2A(1+x) = x + \frac{1}{8}x' + \frac{3}{128} \qquad for\ \frac{1}{2} \le x \le \frac{3}{4}, \tag{3.12}$$

and

$$log_2A(1+x) = x + \frac{1}{4}x' \qquad for\ \frac{3}{4} \le x \le 1. \tag{3.13}$$

where $x'$ is the bit-by-bit binary complement of $x$.

In 1970 Hall [61] proposed a limited piecewise linear approximation Lin2Log curve with the added constraint that the coefficients can be easily calculated. The algorithm used in [61] defines four equal regions for approximating a logarithmic curve. In this algorithm applications to digital filtering computations are considered and log-antilog multiplication is useful for parallel digital filter banks and multiplicative digital filters.

The logarithmic approximation in [61] is defined as

$$log_2(1+x) = x + \frac{37}{128}x + \frac{1}{128} \qquad for\ 0 \le x \le \frac{1}{4}, \tag{3.14}$$

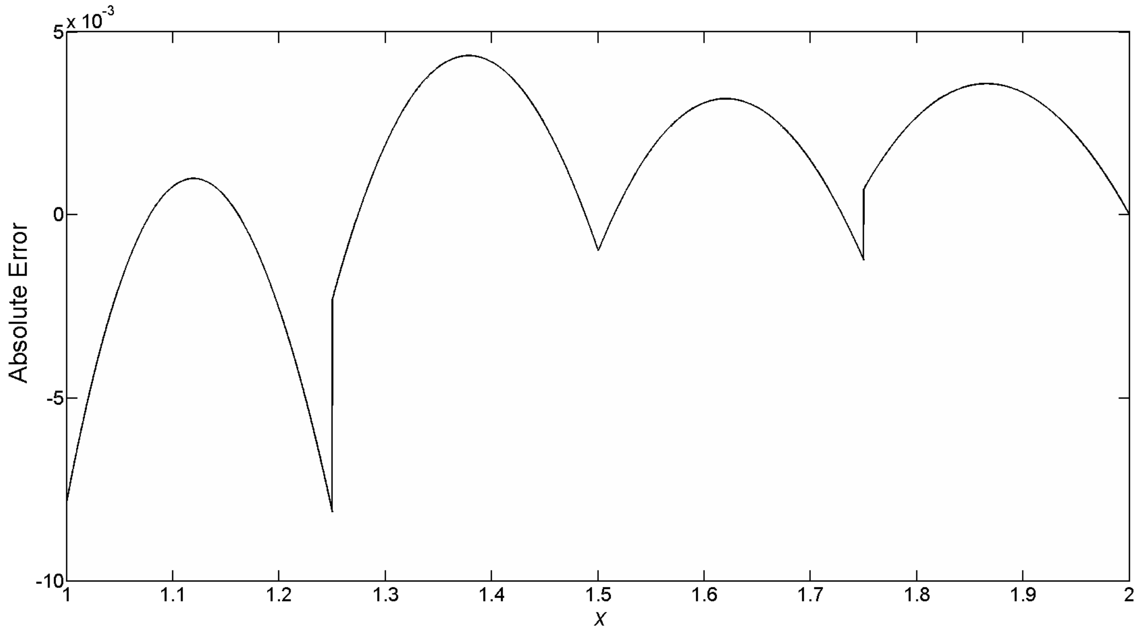
$$log_2(1+x) = x + \frac{3}{64}x + \frac{1}{16} \qquad for\ \frac{1}{4} \le x \le \frac{1}{2}, \tag{3.15}$$

$$log_2A(1+x) = x + \frac{7}{64}(1-x) + \frac{1}{32} \qquad for\ \frac{1}{2} \le x \le \frac{3}{4}, \tag{3.16}$$

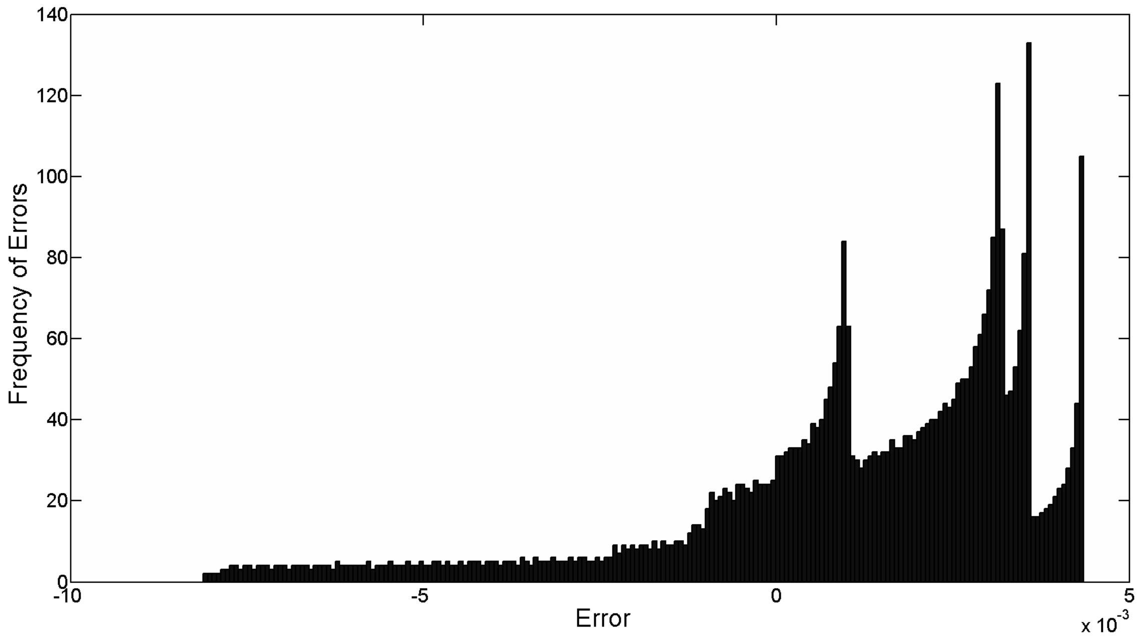
and

$$log_2(1+x) = x + \frac{29}{128}(1-x) \qquad for\ \frac{3}{4} \le x \le 1. \tag{3.17}$$

The absolute error generated by using Hall's logarithmic approximation [61] with respect to original logarithmic value is shown in figure 3.8. Figure 3.8 also shows an unequal error distribution through histogram when using Hall's Logarithmic approximation [61].



(a)



(b)

Figure 3.8: Hall's logarithmic approximation [61] (a) Absolute error (b) Error histogram

Hoefflinger [62–66] showed the implementation of Mitchell's algorithm [59] in VLSI. In these papers logarithmic encoding is by computing the instantaneous value of an $n$, where the signal falls into one of $n$ segments, which are identified by the leading one detector in its binary representation as shown in figure 3.9. The segments are encoded as the $log_2 n$ MSBs of the logarithmic representation. LSBs from the leading one to the bit position following the MSBs in the logarithmic representation are shifted, retaining up to $(m - 1)$ LSBs form bit accuracy over $(n - m)$ octaves.

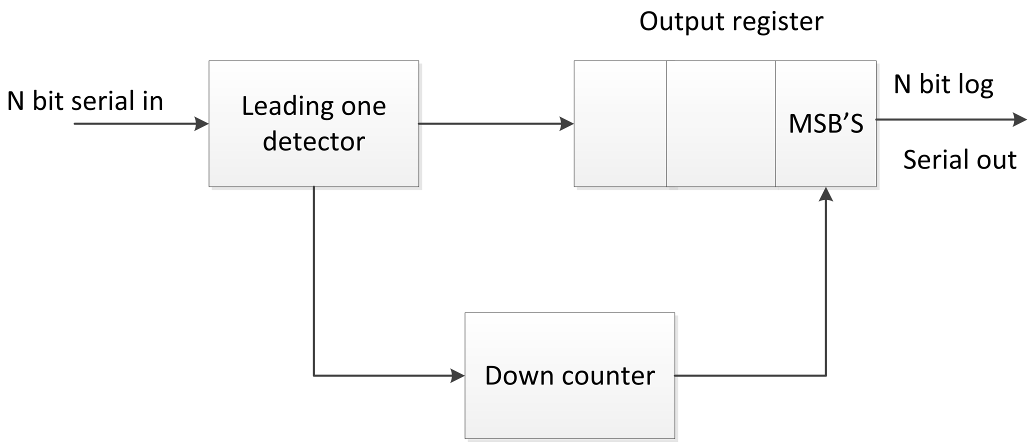
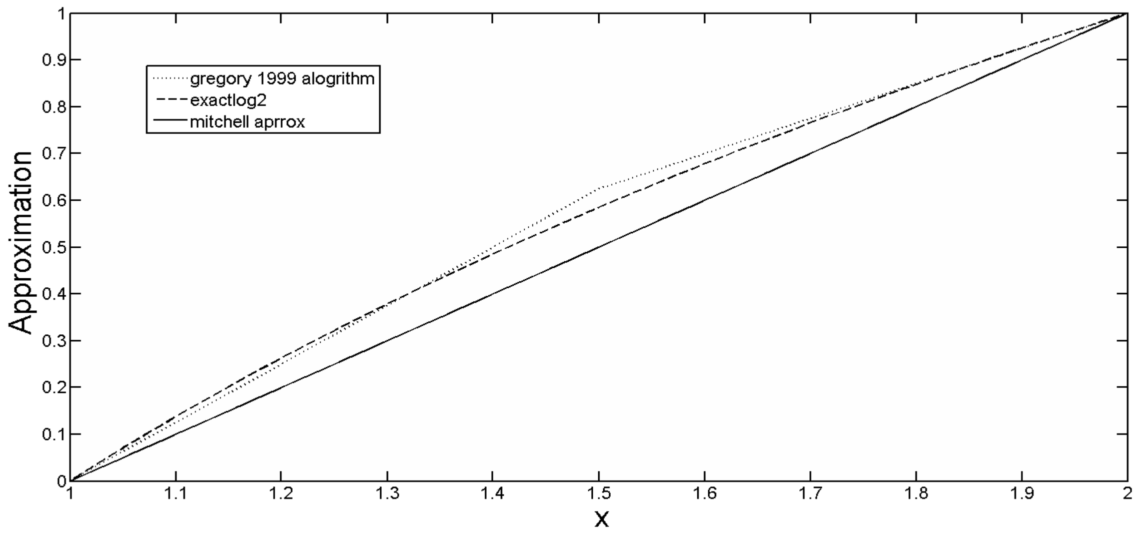


Figure 3.9: Hoefflinger bit-serial logarithmic encoder [62]

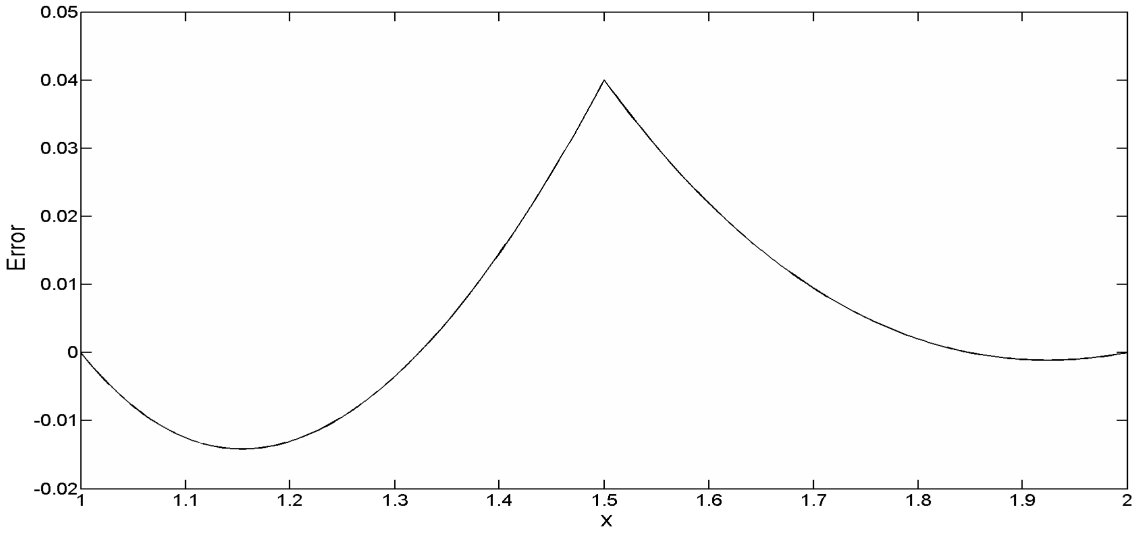
The Hoefflinger also showed that a DIGILOG multiplier. Which is made using above implementation is less than one quarter by size and the multiplication time is one half in comparison with a Booth Wallace multiplier [63]. This implementation is preferred in signal processing operations.

Gregory et al. [67] in 1999 proposed a new algorithm in which approximation is performed using only combinational logic and requires no multiplication. In the Gregory et al. algorithm [67] the error of Mitchell's approximation [59] is reduced by dividing each cycle into two equal regions of different slope. In order to meet the original logarithmic value at mid point of the cycle, the slope in the lower half of the cycle is increased. For values past the midpoint of cycle, the slope is decreased so that the approximate logarithm approaches the exact logarithm value at the end point of the cycle. Increasing and decreasing the slope is achieved by shifting a fractional part of the approximated logarithm value and adding to the original fractional part to reduce errors.

MATLAB simulation graphs are provided in figure 3.10 where the slope of cycle is changed using only first 3 MSBs of the fractional part.



(a)



(b)

Figure 3.10: Gregory 1999 [67] (a)Logarithmic approximation (b) Absolute error

Siferd et al. [56] in 2003 used approximations on Mitchell's algorithm [59]. The improvements required minimal hardware additions, there is a trade-off between accuracy obtained and complexity of correction. 2 region, 3 region and 6 region approximations were made.

The original logarithmic value is

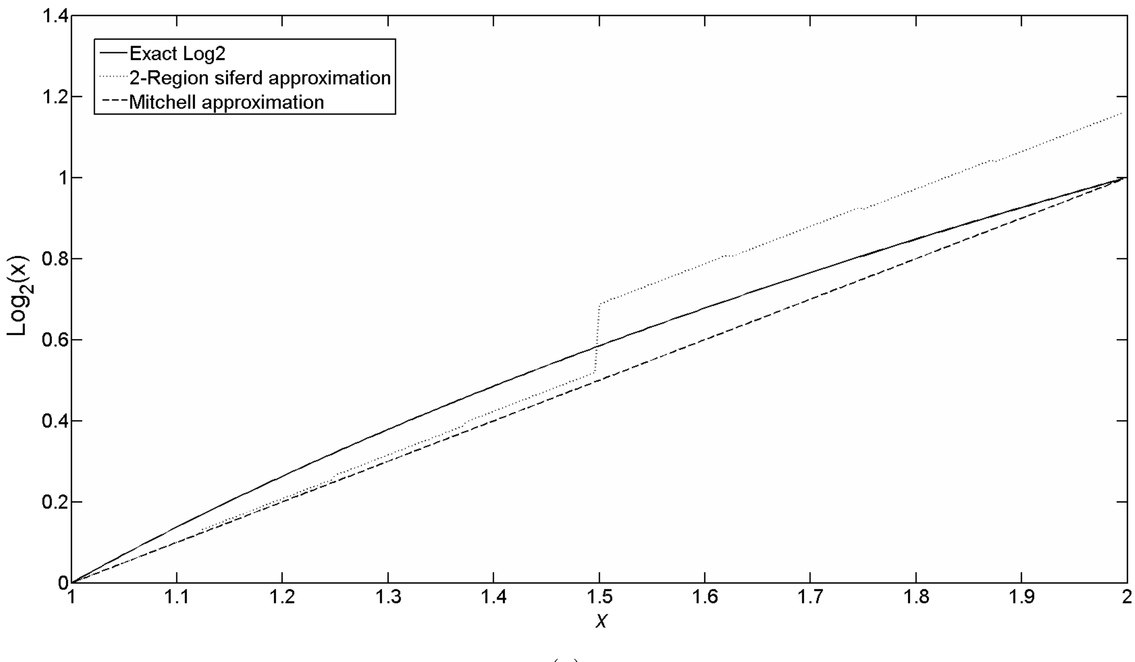
$$N = 2^k(1+m) \tag{3.18}$$

$$log_2 N = k + log_2(1+m) \tag{3.19}$$

For the 2-region correcting algorithm, $log_2(1+m)$ is approximated as $log_2(1+m)'$. The logarithmic approximation and absolute error for 2-region are shown in figure 3.11.

$$log_2(1+m)' = m + \frac{1}{4}(m_{3MSBits}) \qquad for\ 0 \leq m \leq \frac{1}{2} \tag{3.20}$$

$$log_2(1+m)' = m + \frac{1}{4}(1 - m_{3MSBits} - 2^{-3}) \qquad for\ \frac{1}{2} \leq m \leq 1 \tag{3.21}$$

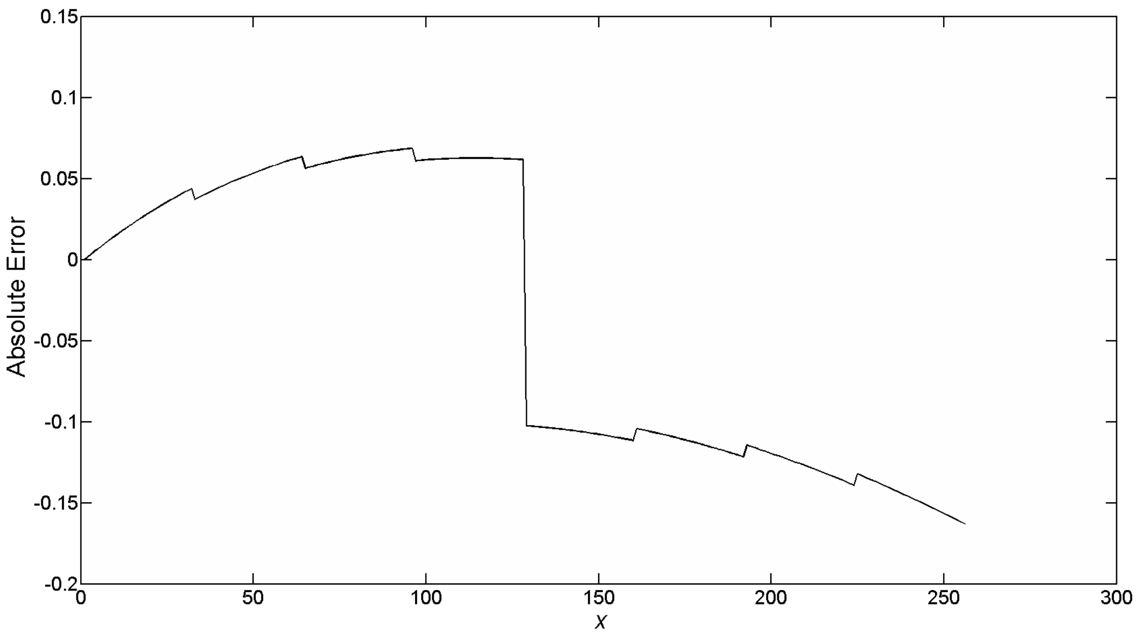


(a)

Figure 3.11: Siferd [56] 2-region (a) Logarithmic approximation

For 3-region approximation

$$log_2(1+m)' = m + \frac{1}{4}(m_{4MSBits}) \qquad for\ 0 \leq m \leq \frac{1}{4} \tag{3.22}$$



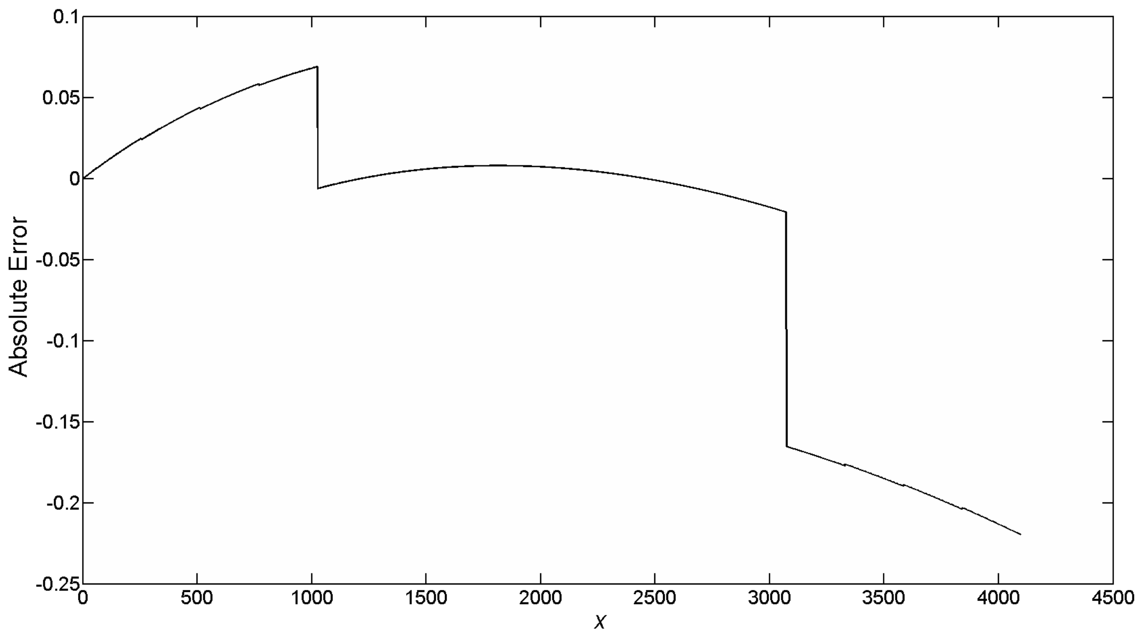
(b)

Figure 3.11: Siferd [56] 2-region (b) Absolute error

$$log_2(1+m)' = m + 2^{-4} + 2^{-6} \qquad for\ \frac{1}{4} \le m \le \frac{3}{4} \tag{3.23}$$

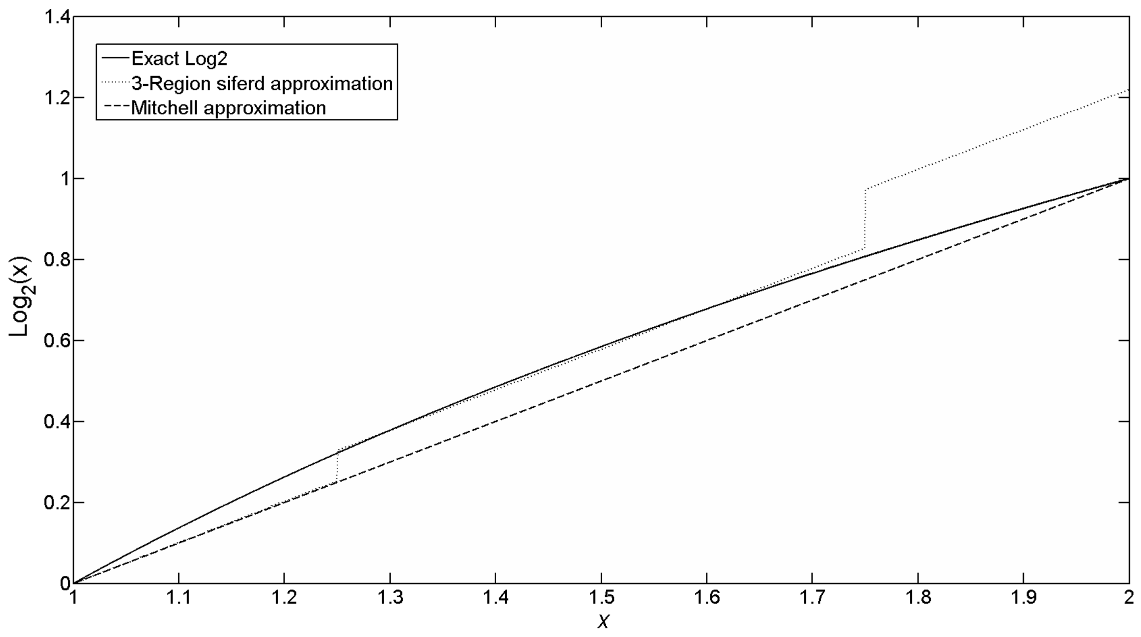
$$log_2(1+m)' = m + \frac{1}{4}(1 - m_{4MSBits} - 2^{-4}) \qquad for\ \frac{3}{4} \le m \le 1 \tag{3.24}$$

The logarithmic approximation and absolute error for 3-region are shown in figure 3.12.



(b)

Figure 3.12: Siferd [56] 3-region (b) Absolute error



(a)

Figure 3.12: Siferd [56] 3-region (a) Logarithmic approximation

For 6-region approximation

$$log_2(1+m)' = m + \frac{1}{4}(m_{6MSBits}) \qquad for\ 0 \leq m \leq \frac{1}{16} \tag{3.25}$$

$$log_2(1+m)' = m + \frac{1}{4}(m_{6MSBits}) + 2^{-6} \qquad for\ \frac{1}{16} \leq m \leq \frac{1}{4} \tag{3.26}$$

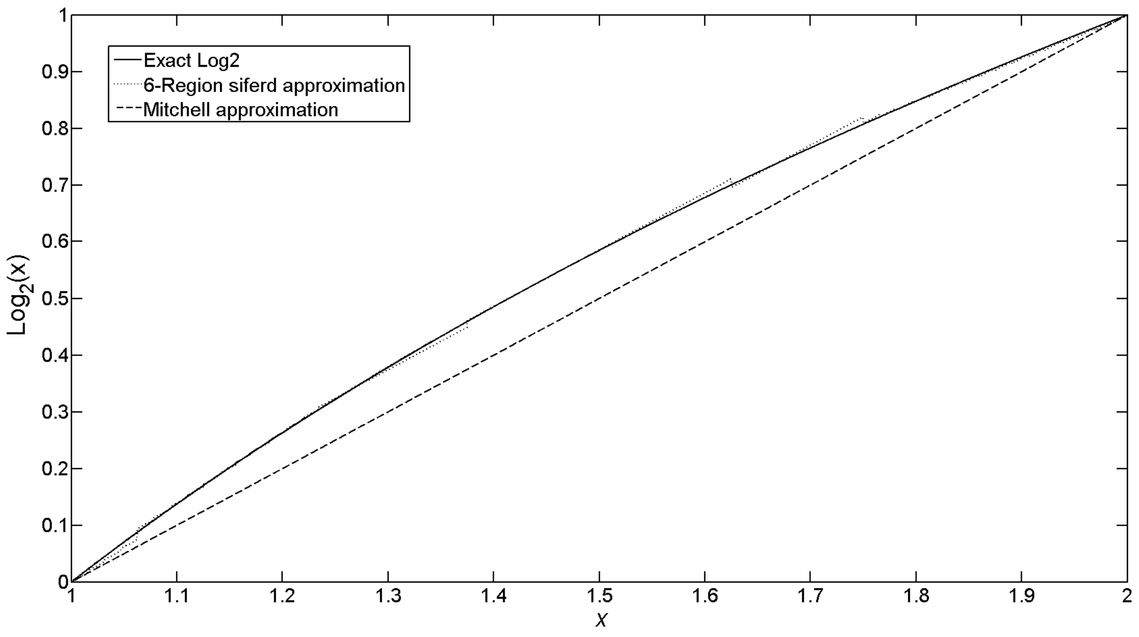
$$log_2(1+m)' = m + 2^{-4} + 2^{-7} + 2^{-8} \qquad for\ \frac{1}{4} \leq m \leq \frac{3}{8} \tag{3.27}$$

$$log_2(1+m)' = m + 2^{-4} + 2^{-6} + 2^{-7} \qquad for\ \frac{3}{8} \leq m \leq \frac{27}{40} \tag{3.28}$$

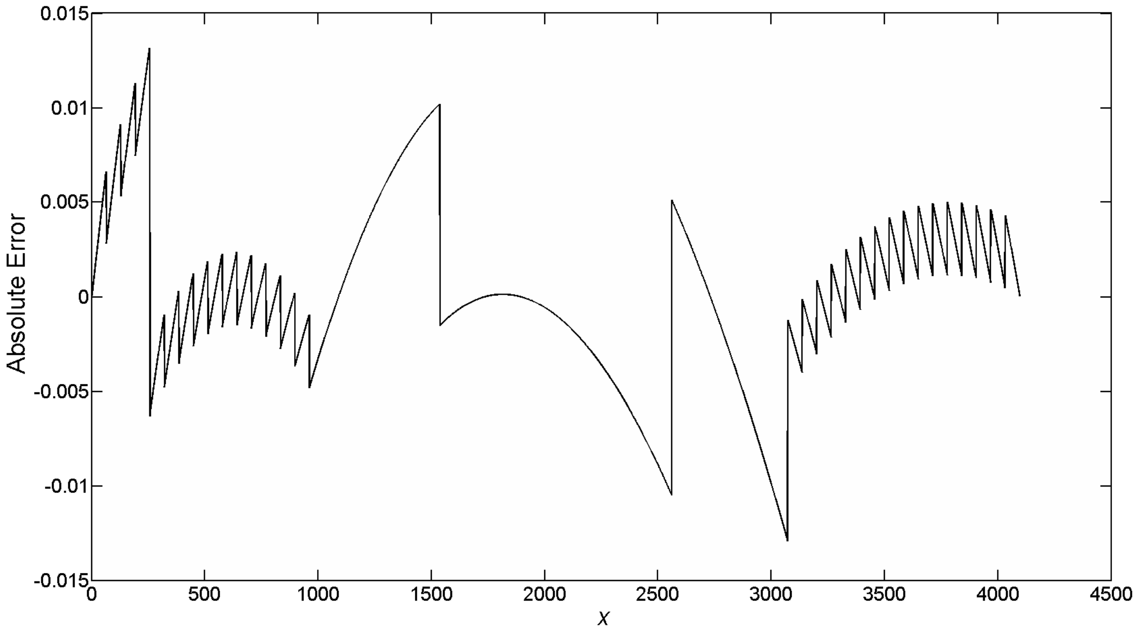
$$log_2(1+m)' = m + 2^{-4} + 2^{-7} \qquad for\ \frac{27}{40} \leq m \leq \frac{3}{4} \tag{3.29}$$

$$log_2(1+m)' = m + \frac{1}{4}(1 - m_{6MSBits} - 2^{-7}) \qquad for\ \frac{3}{4} \leq m \leq 1 \tag{3.30}$$

The logarithmic approximation and absolute error for 6-region are shown in figure 3.13.



(a)



(b)

Figure 3.13: Siferd [56] 6-region (a) Logarithmic approximation (b) Absolute error

### 3.3.2 PWL and LUT Algorithms

This section covers Lin2Log and Log2Lin conversion algorithms using PWL and LUT approximation methods. Brubaker [68] in 1975 proposed such a method to improve Mitchell's approximation by using a look up table (LUT). Brubaker [68] showed a comparison of direct multiplication using a LUT and multiplication using logarithms via a LUT to store their values. Multiplication time for direct multiplication using a LUT required one memory access while logarithmic multiplication required two memory access times plus an addition, which affected the speed of logarithmic multiplication by a factor of one third. The area (number of bits) of the LUT in multiplication via Logarithmic multiplication was smaller in comparison with the number of bits required in LUT via direct multiplication. The accuracy achieved by Brubaker's [68] method was also affected by LUT size. For a given error considerably fewer bits were needed for three LUTs than for a direct multiplication using a single LUT. Brubaker [68] showed that hardware was very well suited for implementing parallel multiplication in applications such as digital filters.

Kmetz [69] in 1986 presented a method to improve Mitchell's algorithm [59]. In Kmetz's [69] algorithm the difference between exact logarithmic value and Mitchell's approximation [59] value is stored in a ROM. Mitchell's approximation [59] value is also used as an address for LUT according to which difference is sent to adder where difference or LUT value is added to the corresponding Mitchell approximation [59] value. Kmetz [69] in his paper is using a normalised floating point number so one is subtracted from the fractional part which is same as Mitchell's approximation [59] value and the exponent of the fractional part is sent to the log storage register, where it is used as a characteristic of the logarithm value and cocatenated with the corrected mantissa value as shown in figure 3.14. The accuracy of Kmetz's approach [69] is dependent on the number of bits used as the address of the error LUT which would be prohibitively large for higher accuracies. A MATLAB simulation for Log approximation using Kmetz's method with a comparison with the original logarithmic curve is shown in figure 3.15 with the absolute error.

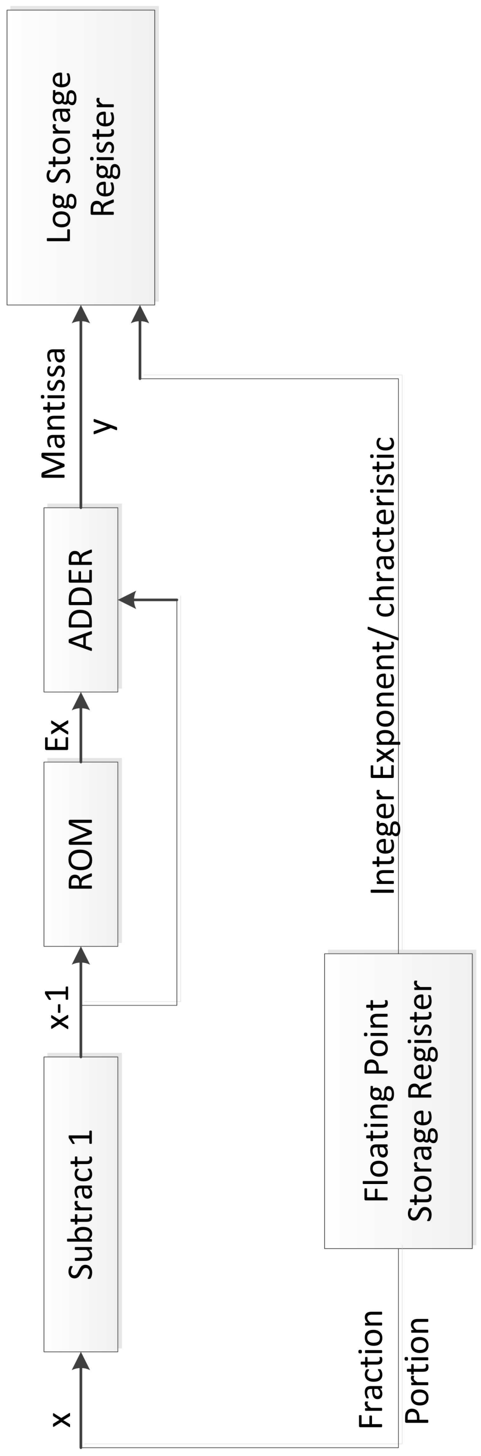
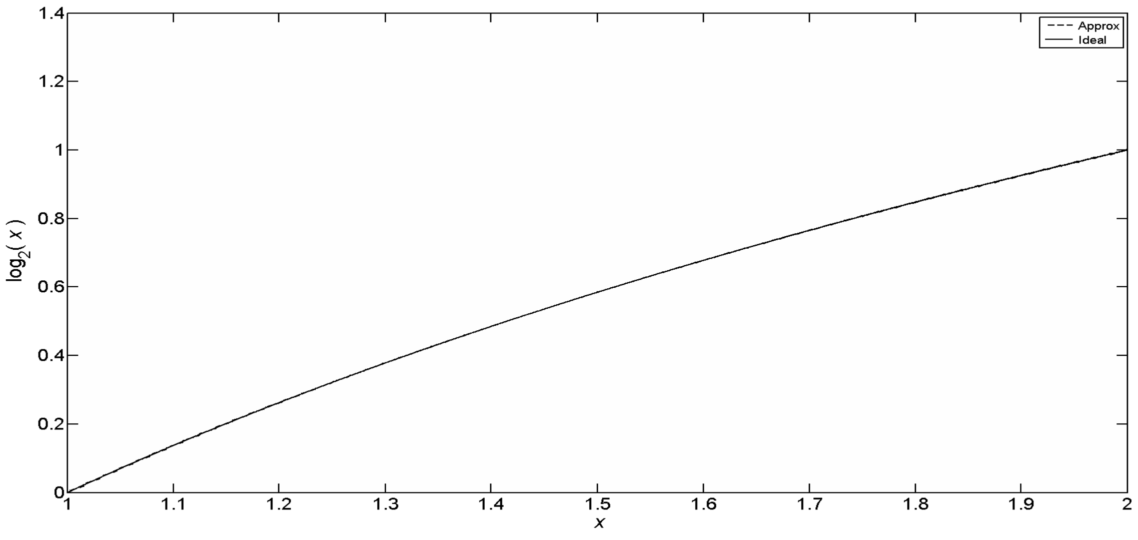
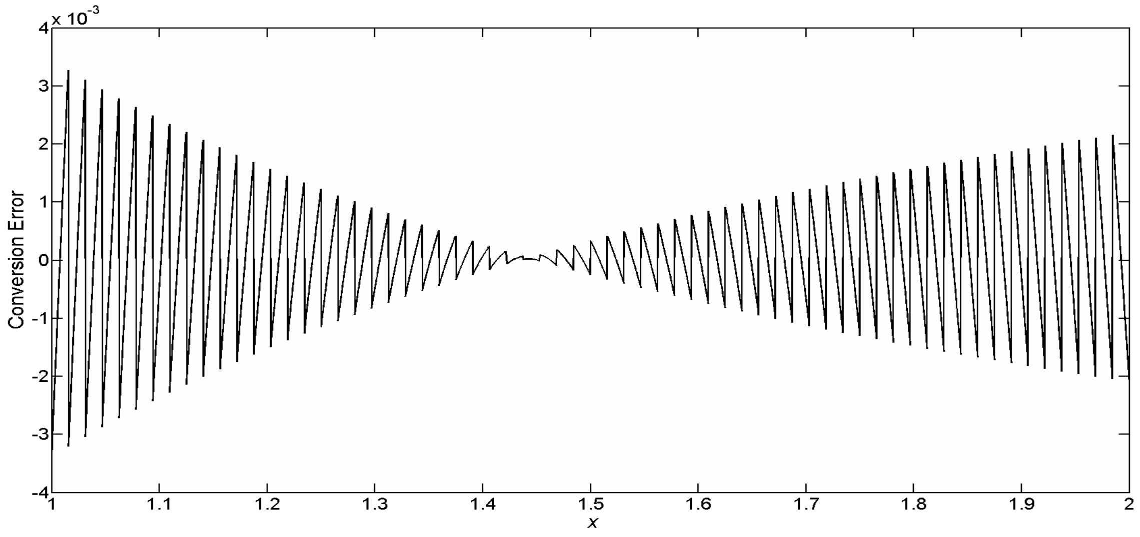


Figure 3.14: Kmetz's [69] principal of proposed approximation.



(a)



(b)

Figure 3.15: Kmetz's [69] (a) Logarithmic approximation (b) Absolute error

Maenner [70] in 1987 proposed a method on the Mitchell's algorithm [59] by using a look up table. Maenner's method [70] can be implemented in software and is therefore well suited for usage in personal computers. The basic idea in Maenner's algorithm was to split a fraction part into two parts. To explain Maenner's algorithm mathematically let Z be a binary fraction so by computations we get

$$Z = f(a_{k-1}, ........, a_{k-m}) + \sum_{i=m+1}^{k} a_{k-i}2^{-i}. \tag{3.31}$$

The function $f$ is given as look up table. For all combinations of $a_{k-1}, ........, a_{k-m}$, the table contains one entry each with a value precomputed to minimise the approximation error. The first step in Maenner's algorithm [70] is to normalise the arguments as done in Mitchell's approximation [59]. The second part is to replace the first $m$ bits of the binary fraction by an element of the look up table. A case of $m = 16$ is shown in figure 3.16 where the total number of bits is 32.

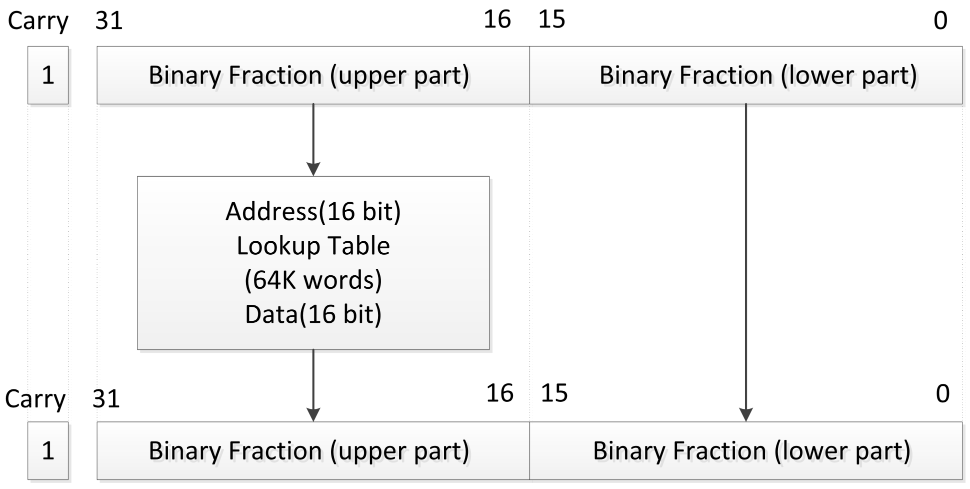
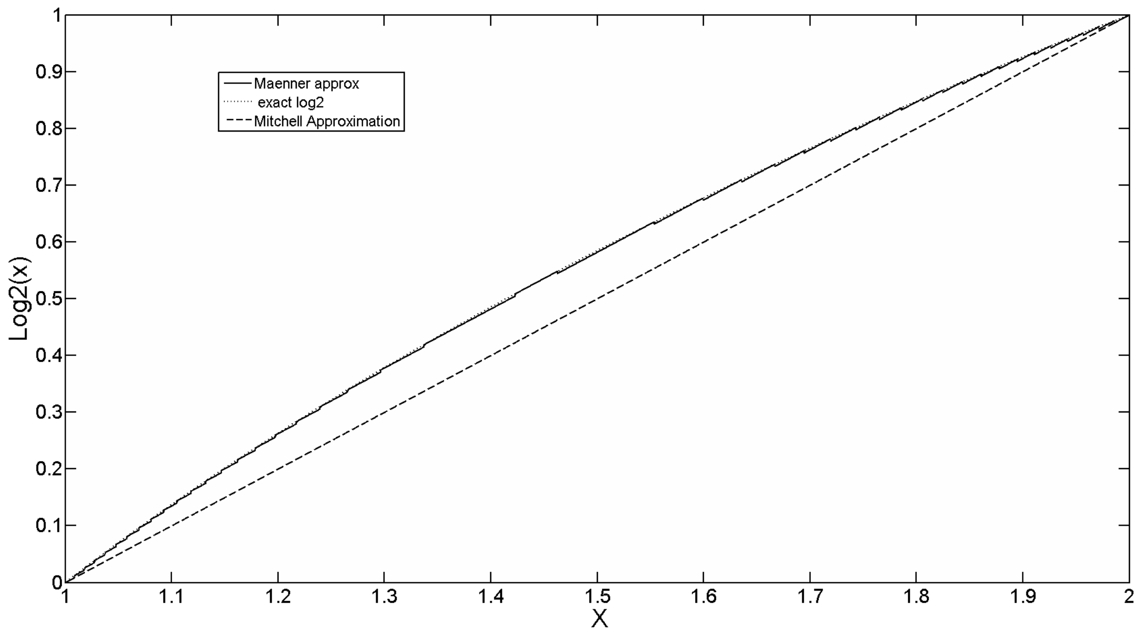

Carry
31
16
15
0
1
Binary Fraction (upper part)
Binary Fraction (lower part)
Address(16 bit)
Lookup Table
(64K words)
Data(16 bit)
Carry
31
16
15
0
1
Binary Fraction (upper part)
Binary Fraction (lower part)

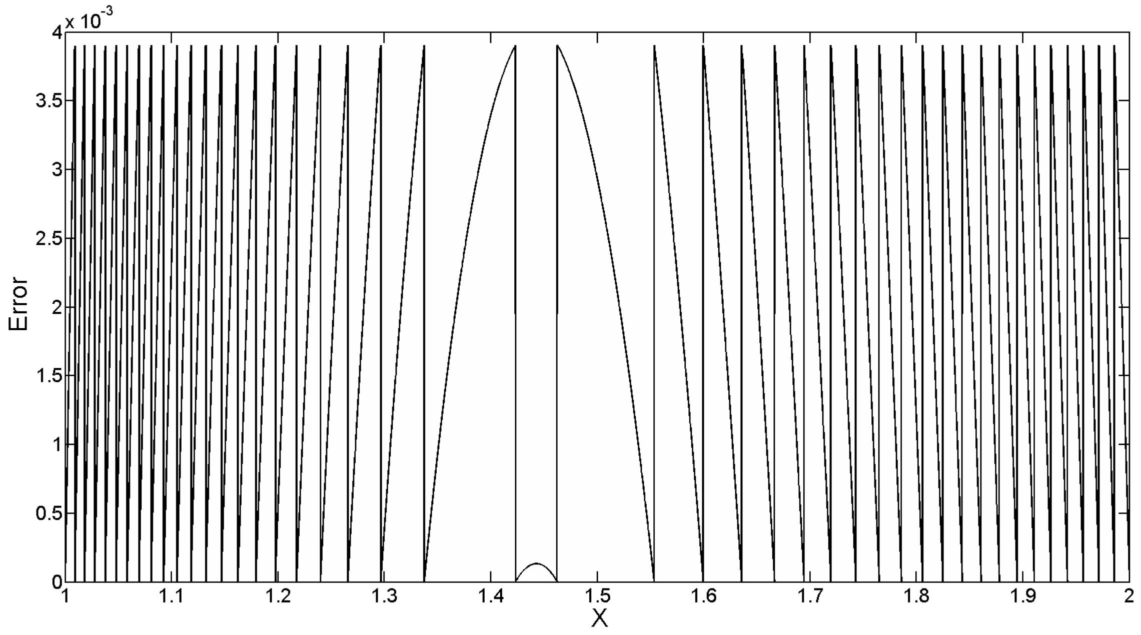
Figure 3.16: Maenner [70] principal of proposed approximation.

According to Maenner [70] using the first $m$ bits of the fraction for a look up table replaces $m$ bits by their exact precomputed values, so the error is restricted in bits from $m + 1$ down to zero. He presented his algorithm on a 68000 microprocessor, using a 64k word look up table and the approximation error was in the order of magnitude of $10^{-6}$, showing a factor of 100 or 1000 times smaller than the errors obtained with earlier approximations.

A simulation of Maenner's approximation [70] is shown in figure 3.17 where the total number of bits is taken as 16. A logarithmic approximation and absolute error is calculated.



(a)



(b)

Figure 3.17: Maenner [70] (a) Logarithmic approximation (b) Absolute error

Maenner [70] claimed that by using $m$ segments the error in the conversion is reduced by a factor of $2m$. However in 2003 Arnold published a paper [71] in which he made a detailed examination of this algorithm and concluded that it could not achieve the accuracy claimed in the original paper because, although similar, the error curve is not the same in each segment. Maenner's algorithm [70] uses the same interpolation throughout the segments and assumed the error at the endpoints was zero. This was not the case pointed out by Arnold and hence that the conversion error was substantially larger than Maenner [70] had claimed.

### 3.3.3 Polynomial Approximation Methods

This section includes piece-wise polynomial approximation methods to convert a fixed/floating point number to a logarithmic number and vice versa. Marino [72] in 1972 proposed a simple circuit for approximating the Mitchell [59] error curve. This was achieved by using two quadratic approximations to the curve. The multiplicative factors of the quadratic terms were chosen to minimise the arithmetic overhead required which was an expensive resource at the time. Marino [72] claimed an improvement, by a factor of 2.5 in the absolute maximum error over Mitchell's approximation [59]

The error of logarithmic expression is written as

$$E_{log_2}(x) = log_2(N) - log_2(N)' = log_2(1+x) - x \tag{3.32}$$

In Marino's [72] approximation $E_{log_2}(x')$ is an approximate fit of $E_{log_2}(x)$ which is calculated using a method called divided difference giving the expression

$$E_{log_2}(x') = 4t(x - x^2) \tag{3.33}$$

which is calculated by

$$E_{log_2}(x') = (x - x_a)f(x_a, x_b) + (x - x_a)(x - x_b)f(x_a, x_b, x_c) \tag{3.34}$$

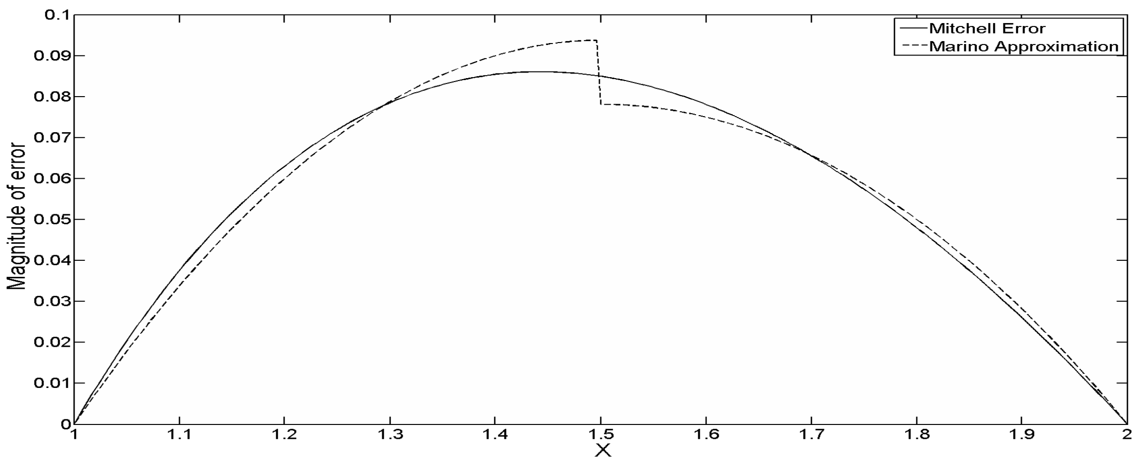
where $x_a = 0$, $x_b = 0.5$ and $x_c = 1$ and $f(x_b) = E_{log_2}(x'_b) = t$ $f(x_a) = 0$ $f(x_a, x_b) = 2t$ $f(x_a, x_b, x_c) = -4t$.

Here $t$ represents the ordinate of the parabola, because of asymmetry in Mitchell [59] error curve there are two approximations. The curve is divided into two halves as follows

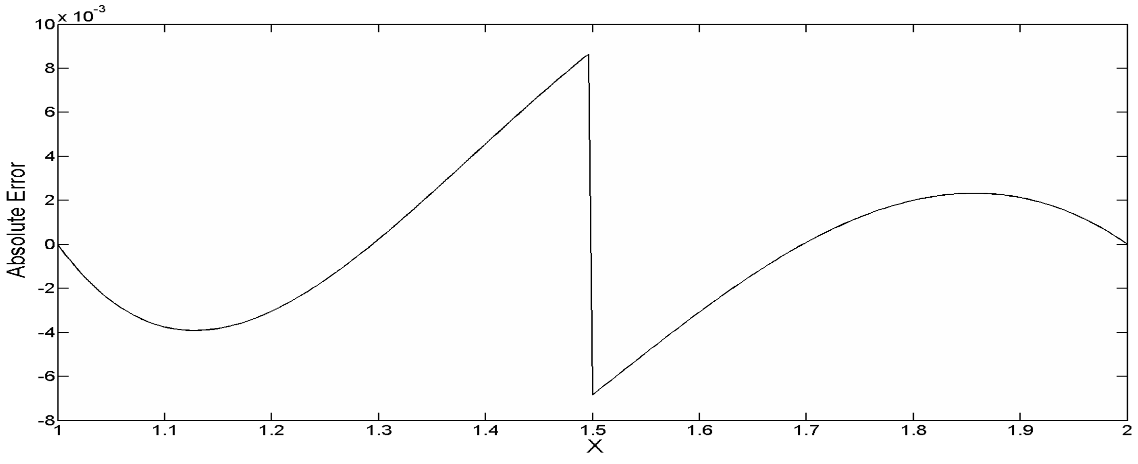
$$E_{log_2}(x') = \frac{4t1}{0.75}(x - x^2) \qquad for x < 0.5 \tag{3.35}$$

$$E_{log_2}(x') = \frac{4t2}{0.75}(x - x^2) \qquad for x \geq 0.5 \tag{3.36}$$

Marino [72] defines an approximation for the $x^2$ function. The maximum error in Marino's approximation [72] occurs at $x = 0.5$ as +0.0029 and -0.0020 for the first and second curve respectively and are shown in figure 3.18.



(a)



(b)

Figure 3.18: Marino [72] (a)Logarithmic approximation (b) Absolute error

Mori [73] in 1987 used a general non uniform piecewise polynomial approximation on the Mitchell [59] error curve. The Mitchell [59] error curve in is defined as

$$E_{log_2}(x) = log_2(1 + x) - x. \tag{3.37}$$

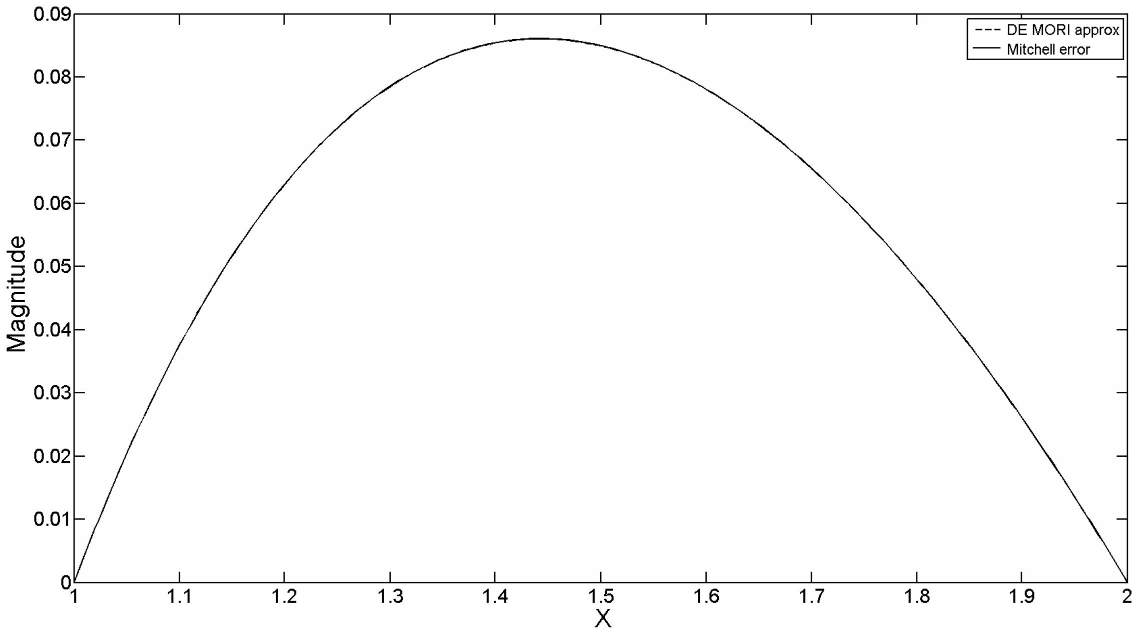
Mori [73] used a polynomial fitting function of the type

$$Z_i = A_i x^2 + B_i x + C_I + x \qquad 1 \leq i \leq I. \tag{3.38}$$

where in each interval $[X_{i-1}, X_1]$ error curve function is

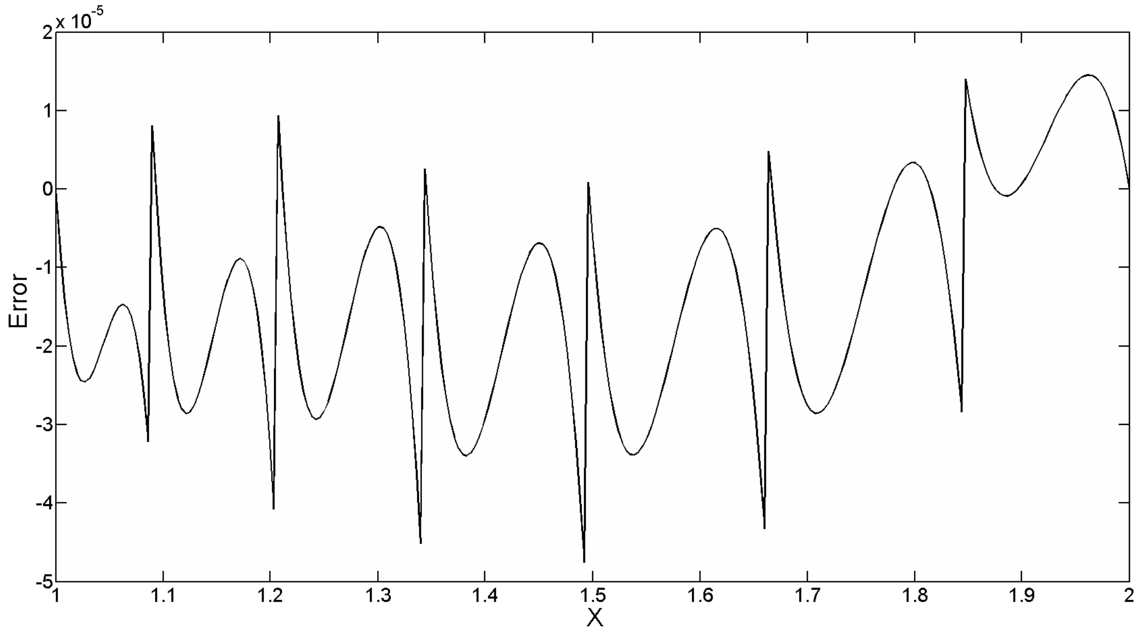
$$Q_i = A_i x^2 + B_i x + C_i \tag{3.39}$$

This error curve function is used to fit the Mitchell [59] error curve. The MATLAB simulations for Mori approximations [73] are shown in figure 3.19. The Mori approximation [73] uses ROM for values of coefficients $A$, $B$ and $C$ in polynomial equations.



(a)

Figure 3.19: Mori [73] (a) Logarithmic approximation



(b)

Figure 3.19: Mori [73] (b) Absolute error

Lewis [74] describes a new method for polynomial interpolation in hardware, with advantages demonstrated by its application to an accurate logarithmic number system (LNS) arithmetic unit. The algorithm described by Lewis [74] uses interleaved memory for interpolating by storing actual value of function ($f(x)$) instead of storing their coefficient values, as done by previous algorithms. The use of an interleaved memory is that it reduces storage requirements by allowing each stored function value to be used in interpolation across several segments. This strategy is shown to be always used for fewer words of memory than an optimised polynomial with stored polynomial coefficients.

The stored function value interpolator uses a slightly different approach for interpolation intervals. 91k bits of ROM are used in implementing the second order polynomial interpolator in the LNS arithmetic unit. Lewis's method [74] claims many different accuracies required for the LNS unit are possible and are better and less complex than the previous LNS unit.

In computer graphics algorithms normal operations like division, multiplication, square etc. are frequently performed but are very slow if done by software or very expensive if done by hardware. Knittel [75] in 1994 presented an algorithm in which operands are converted into logarithms, turning division into subtraction and exponentiations into multiplications.

The logarithm in Knittel's algorithm [75] is calculated using a piecewise linear approximation of the form

$$g = m_i x + b_i \tag{3.40}$$

By storing the coefficients $b_i, m_i$ and use of ternary encoding an $n$-digit binary number can be expressed as $n + 1$ digit ternary number. In this way, the set of straight lines which can be constructed using a fixed number of adders is enlarged significantly. Knittel in his paper describes an algorithm to calculate distance between two adjacent segments on the curve when using linear approximation (see figure 3.20). All segments which do not increase the error if dismissed are removed. When this distance between approximation and original curve increases above the required accuracy a new segment starts to minimise error function. Using this algorithm there is a significant reduction in memory, which is used to store the coefficient values for approximation.

The patents [76–85] published by S.Pan and S.T. Wang describe a second order interpolator defined as

$$y = cx^2 + bx + a. \tag{3.41}$$

Out of 23 factional bits of floating point number the most significant 9 bits are used as an address for 2 LUT ROM storing coefficients of zero and first order.

The second order term is encoded using a LUT ROM whose address of 9 bits is made of 4 most significant bits of zero and first order address bits and 5 bits of the input that have not been used to address the zero and first order coefficients. Values of the second order term have been empirically determined by observing the variance of the second order term removing the need for a multiplier. The patent does not include details about overall accuracy achieved and the circuit uses 25k

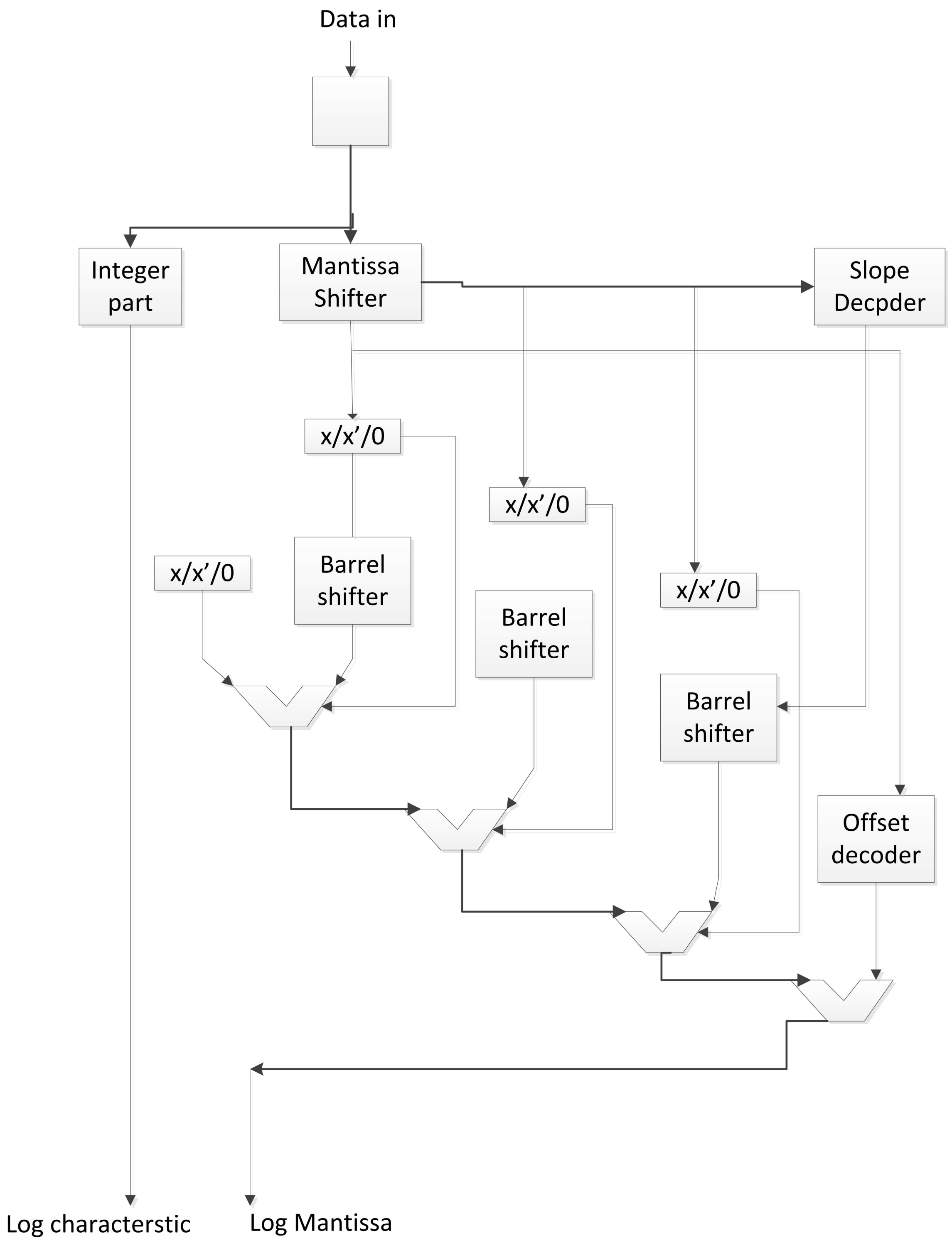


Figure 3.20: Knittel [75] non uniform log approximation.

bits of total memory. The architecture has been used and modified in many papers and patents by S. Pan and S.T. Wang. The general architecture of the second order interpolator is shown in figure 3.21.

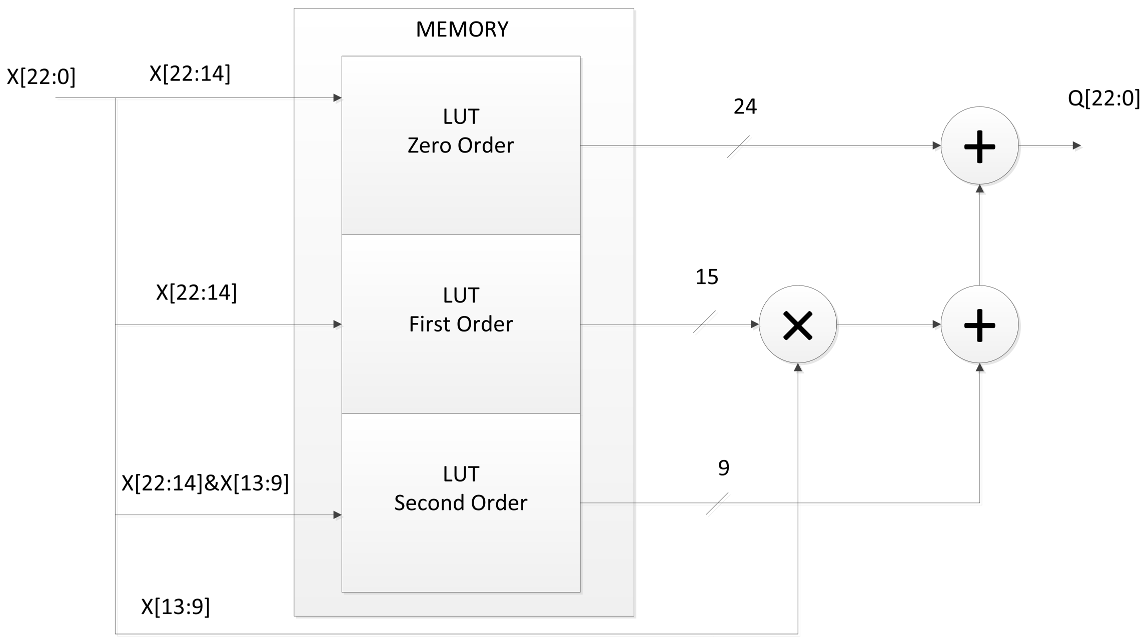


Figure 3.21: S.Pan [79] Lin2Log architecture.

## 3.3.4 CORDIC and Taylor algorithms

This section includes approximation methods to convert a fixed/floating point number to a logarithmic number and vice versa based on some well known mathematics techniques such as the Taylor series, CORDIC methods etc.

The CORDIC (Coordinate Rotation Digital Computer) algorithm is used to calculate hyperbolic and trigonometric functions. This algorithm can be implemented through a software or hardware approach. However, it is suitable in hardware implementation as it uses only adders, shifters and LUT. The CORDIC algorithm for curve fitting techniques such as Lin2Log approximation etc has been used in [86,87].

[86] in 2013 proposed a CORDIC algorithm-based logarithmic converter. The logarithmic converter [86] supports the logarithmic transformation of data with a number of bits up to 48. The algorithm proposed in [86] uses the CORDIC IP core of Xilinx ISE to calculate the inverse hyperbolic tangent and natural logarithm. The algorithm [86] implementation in hardware is performed by using standard Verilog hardware description language in ModelSim PE and the designed circuit is mapped

onto a Xilinx xc5vsx95t device.

[87] in 2009 presented a new decimal floating-point CORDIC algorithm for the computation of transcendental functions. Vazquez et al. [87] proposed a novel coding scheme to use a unified algorithm for both circular and hyperbolic coordinates in comparison to standard binary CORDIC algorithms using a constant scale factor. Vazquez et al. [87] compares his new decimal floating-point CORDIC algorithm with the LUT methods and shows a significant reduction of latency and storage.

However, the CORDIC algorithm has been challenged in the past by many other approximation methods such as Chen et al. [88] who in 2012 proposed an algorithm and architecture of the decimal floating-point logarithmic converter, based on the digit-recurrence algorithm with selection by rounding. The proposed algorithm in [88] showed a latency 3.88 times faster than that of the unit based on the CORDIC algorithm in [87].

Mansour et al. [89] in 2015 presented a new method called Floor Shift based on [90] for fast logarithm conversions. Mansour et al. [89] also combine the Floor Shift algorithm with the Taylor series to improve the accuracy of the output. The proposed method is compared with other existing common algorithms such as the CORDIC and LUT-based approximations. Mansour et al. [89] shows that Taylor-based approximation is the most power based efficient design, maintaining similar accuracy with LUT-based approximations with a reasonable latency.

A Taylor series used in logarithmic approximation methods is defined as a representation of a function of an infinite sum of terms that are calculated from the values of the function's derivatives at a single point. A Taylor series of function $f(x)$ mathematically can be represented as

$$f(x) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + .... \tag{3.42}$$

$$= \sum_{n=0}^{\infty} \frac{f^n(a)}{n!}(x-a)^n \tag{3.43}$$

where $a$ denotes the real/complex number, $n!$ denotes the factorial of $n$ and $f^n(a)$ denotes the $n^{th}$ derivative of $f$ evaluated at the point $a$.

For linear interpolation the Taylor coefficients correspond to a tangent line through the tabulated point. A truncated Taylor series can achieve a significant improvement in accuracy, requiring only one or more LUTs to store the approximation coefficients and, in most cases, a multiplier. A truncated Taylor series for approximating functions required by logarithmic number system (LNS) with more accuracy than linear interpolation while using only a single multiplication was proposed by Arnold [91] in 2001. Arnold's method [91] uses two ROMs to give accuracy of quadratic interpolation, whilst the other method uses one ROM to give four to six bits better accuracy than linear interpolation. In [91] Arnold starts with explaining few existing interpolation methods with their limitations and overcoming those limitations he proposes a novel single multiplier quadratic secant multiplier. Some of the interpolation described are

- Linear Tangent (LT) Interpolation

$$f(-n\triangle - \delta) \approx f(-n\triangle) - \delta.D(-n\triangle) \tag{3.44}$$

This method is also known as a Taylor interpolation. Like most linear interpolation techniques [92, 93] it requires one multiplication in hardware. Equation 3.44 in hardware is represented as shown in figure 3.22, where ROM1 represents $f(-n\triangle)$ and ROM2 this $D(-n\triangle)$.

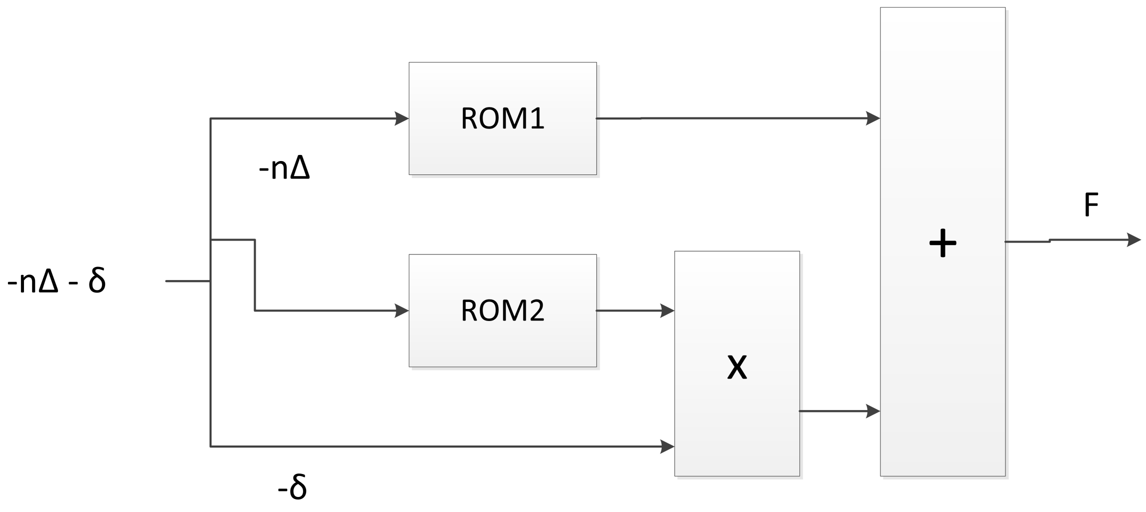


Figure 3.22: Multiplier-based Linear Interpolator [91]

- Multiplierless LT Interpolation

By implementing the interpolation with logarithmic arithmetic instead of fixed-point arithmetic, the multiplication can be eliminated

$$f(-n\triangle - \delta) \approx f(-n\triangle) - b^{log_b(\delta)+log_b(D(-n\triangle))} \tag{3.45}$$

Equation 3.45 in hardware is represented as shown in figure 3.23, where ROM3 represents $log_b(D(-n\triangle))$ , ROM4 $log_b(\delta)$ this and ROM5 is used for anti-log.

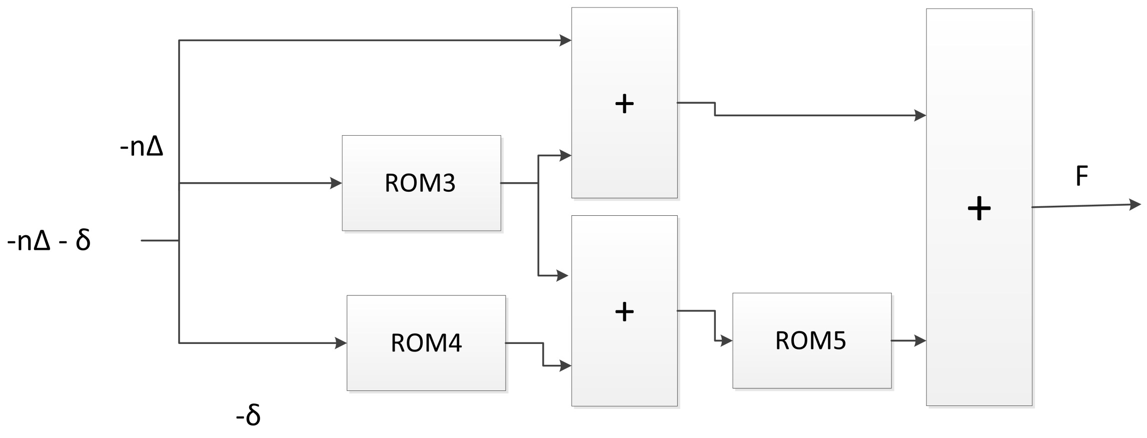


Figure 3.23: Multiplierless Linear Interpolator [91]

• Quadratic Tangent (QT) Interpolation

Coleman [94] extended the idea of tangent-line linear interpolation to quadratic interpolation using an error correcting term, $E(n).P(\delta)$.

$$f(-n\triangle - \delta) \approx f(-n\triangle) - \delta.D(-n\triangle) + E(n).P(\delta) \tag{3.46}$$

Coleman's technique is affordable because only the high-order bits of $\delta$ are required to obtain a satisfactory approximation for P($\delta$), where it requires 4 rom and 2 multipliers.

• Novel single Multiplier QT Interpolation

$$f(-n\triangle - \delta) \approx f(-n\triangle) - \delta.D(-n\triangle) + b^{log_b(\delta)+log_b(D(-n\triangle))} \tag{3.47}$$

• Quadratic Secant (QS) Interpolation

$$f(-n\triangle-\delta)\approx f(-n\triangle)+\delta\frac{f((-n-1)\triangle)-f(-n\triangle)}{\triangle}$$
$$+2(f(-n\triangle)+f(-n-1)\triangle)-2f((-n-0.5)\triangle).(\frac{\delta^2}{\triangle^2}-\frac{\delta}{\triangle}) \qquad (3.48)$$

- Novel Single-Multiplier QS Interpolation

Arnold proposes this method by eliminating two multiplications from the above equation by using logarithmic arithmetic. Hardware realisation is shown in figure 3.24, where ROM6 is $-log_b(D'(-n\triangle))+log_b(ln(b))$, Rom7 is $log_b(\delta\triangle-\delta^2)-1+2log_b(ln(b))$, ROM8 is $(f((-n-1)\triangle)-f(-n\triangle))/\triangle$ and ROM9 is used for antilogarithm function.

$$f(-n\triangle-\delta)\approx f(-n\triangle)+\delta\frac{f((-n-1)\triangle)-f(-n\triangle)}{\triangle}+$$
$$b^{(-n\triangle-2f(-n\triangle)+log_b(lnb)+log_b(\delta\triangle-\delta^2)-1)} \qquad (3.49)$$

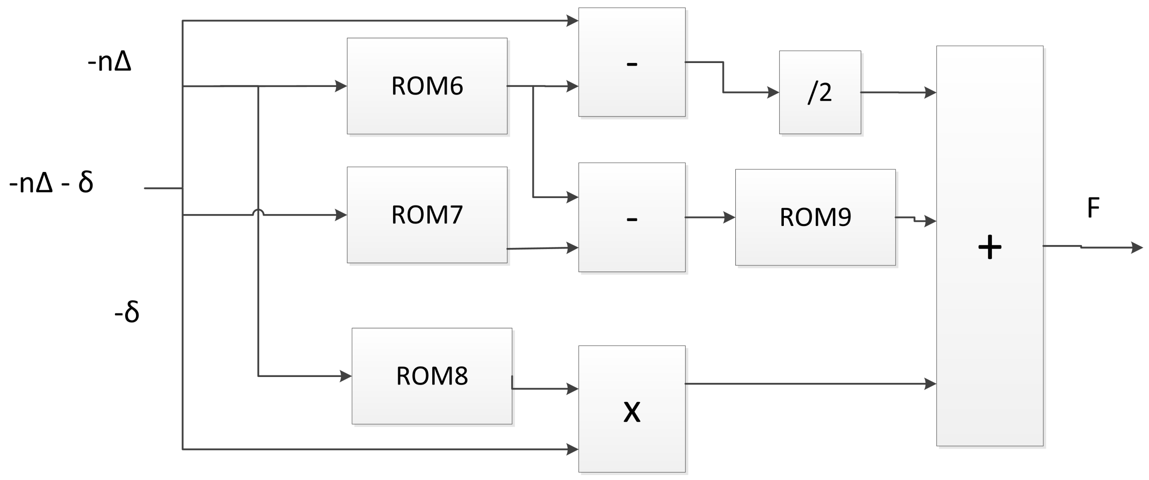


Figure 3.24: Novel Single Multiplier Quadratic Interpolator [91]

### 3.3.5 Other Methods

This section includes conversion of Lin2Log and Log2Lin using some uncommon/-complex methods. One of these is the method proposed by Lewis in [95, 96]. Lewis in [95,96] describes the architecture for performing addition and subtraction of numbers in the logarithmic number system using small lookup tables. Lewis [95, 96] describes a new algorithm for linear approximation using different-sized approximation intervals in each of a number of segments used. A second technique is also proposed using non-linear compression for further reduction in table space by storing the difference between the exact value of the function and a linear approximation. The method describe in [95, 96] allows implementation of logarithmic arithmetic using much less ROM than previously required, making high speed logarithmic arithmetic possible in an area comparable to single-precision floating-point processors.

The method described in [95,96] (see figure 3.25) can be represented mathematically by assuming $a$ and $b$ to be the two numbers represented in LNS (Linear number System) and $c$ to be the sum of these two numbers.

$$c = a + b \tag{3.50}$$

Taking log on both sides

$$logc = log(a + b) \tag{3.51}$$

$$logc = log(a * (1 + \frac{b}{a})) \tag{3.52}$$

$$logc = log(a) + log(1 + \frac{b}{a}) \tag{3.53}$$

$$logc = log(a) + log(1 + 2^{log(b)-log(a)}) \tag{3.54}$$

$$e_c = e_a + f_a(e_b - e_a) \tag{3.55}$$

where

$$f_a(r) = log(1 + 2^r) \tag{3.56}$$

Subtraction between $a$ and $b$ the two input numbers can also take place, assuming $c$ as the result.

$$c = a - b \tag{3.57}$$

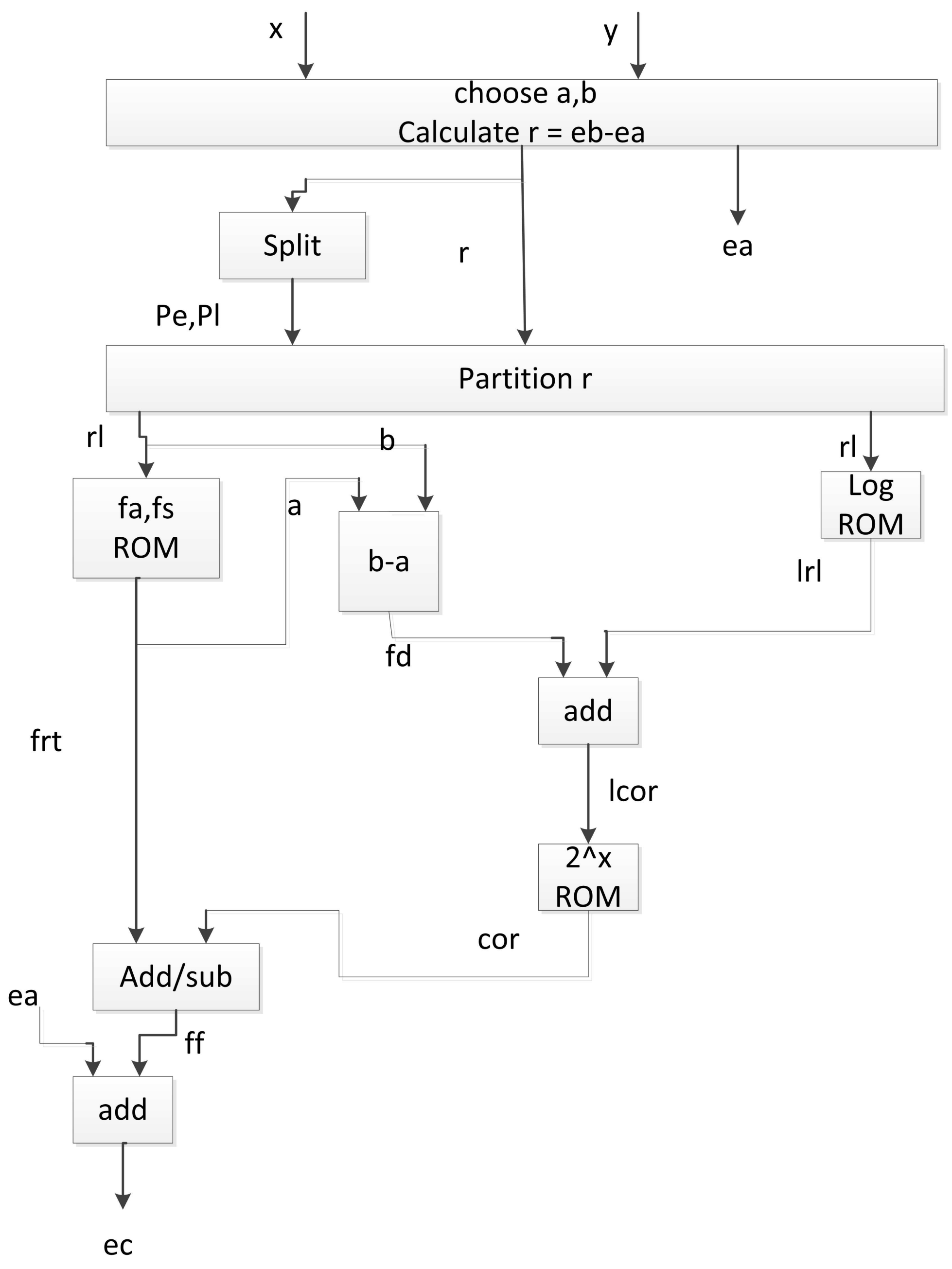


Figure 3.25: Data path for Lewis [96] segmented linear approximation.

$$e_c = e_a + f_s(e_b - e_a) \tag{3.58}$$

where

$$f_s(r) = log(1 - 2^r) \tag{3.59}$$

In linear approximation Taylor's first order equation is

$$f'(x+\triangle x) = f(x) + \frac{\delta f(x)}{\delta x} * \triangle x \tag{3.60}$$

Hence, after computation we get $f_a$ and $f_s$ function in this order

$$f'(x+\triangle x) = f_a(x) + sgn(\triangle x) * exp(log|x| + x - f_a(x)) \tag{3.61}$$

$$f'(x+\triangle x) = f_a(x) - sgn(\triangle x) * exp(log|x| + x - f_s(x)) \tag{3.62}$$

In the second technique proposed in [95, 96], the table compression is used to reduce the size of each of the lookup tables. Nonlinear compression uses the observation that a linear approximation provides a close, but inexact, approximation of the function (see figure 3.26). Since the approximation is close, a table storing the difference between the linear approximation and exact function can use a few bits to represent the difference

$$x = x_b + x_e \tag{3.63}$$

$$f(x) = f(x_b) + \frac{\delta f(x_b)}{\delta x} \times x_e + f\delta(x) \tag{3.64}$$

$$f\delta(x) = f(x) - f(x_b) - \frac{\delta f(x_b)}{\delta x} \times x_e \tag{3.65}$$

Value stored by $\frac{\delta f(x_b)}{\delta x}$ is chosen to optimise the accuracy of the linear approximation and thus minimise the number of bits of ROM.

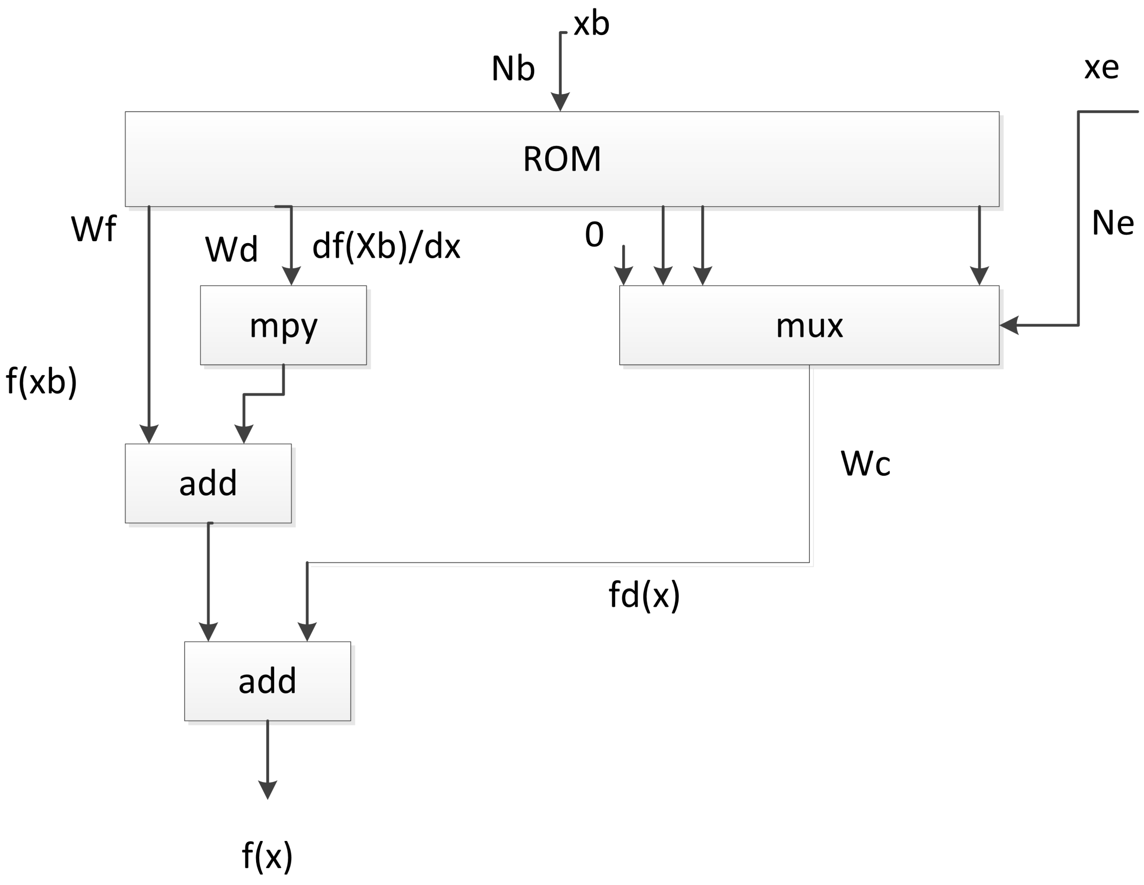


Figure 3.26: Lewis [96] Nonlinear compression.

Huang et al. in 1993, 1994, 1998 and 1999 described and used the algorithm proposed in [97–102] for logarithmic conversion. Huang et al. starts describing the algorithm mathematically by assuming input data $1.x$ can be represented as $2^{0.y}$ (binary exponent).

$$1.x_0.x_1.x_2..........x_{22} = 2^{0.y0.y1..........y22} \tag{3.66}$$

$$\begin{aligned}1.x_0.x_1.x_2..........x_{22} &= 2^{0.y0.y1..........y10} \\ &* 2^{00000000000y11y12...........y22}\end{aligned} \tag{3.67}$$

$$\begin{aligned}(1.x_0.x_1.x_2..........x_{22}) \ * \ 2^{-0.y0.y1..........y10} \\ = 2^{00000000000y11y12...........y22}\end{aligned} \tag{3.68}$$

The left hand side can be written as

$$\begin{gathered}(1.x_0.x_1.x_2..........x_{12}) \ * \ 2^{-0.y0.y1..........y10} + \\ (00000000000x13x14...........x22) \ * \ 2^{-0.y0.y1..........y10}\end{gathered} \tag{3.69}$$

The first term is stored in ROM or PLA and the second term is simplified as

$$= (00000000000x13x14...........x22) \ * \ 2^{-0.y0.y1..........y10} \tag{3.70}$$

Taking log and antilog

$$\begin{aligned} &= 2^{log_2((00000000000x13x14...........x22) \ * \ 2^{-0.y0.y1..........y10})} \\ &= 2^{log_2(00000000000x13x14...........x22) \ + \ log_2(2^{-0.y0.y1..........y10})} \\ &= 2^{(log_2(0.x_{13}.x_{14}....x_{22})+(-13)+(-0.y0.y1.....y10))} \\ &= 2^{(log_2(0.x_{13}.x_{14}....x_{22})+(-0.y0.y1.....y10))} + 2^{-13}\end{aligned} \tag{3.71}$$

The term $2^{(log_2(0.x_{13}.x_{14}....x_{22}+(-0.y0.y1.....y10))}$ can be transformed further and be represented by $2^{-I.F}$. First term is stored in PLA and second term in ROM as shown in figure 3.27.

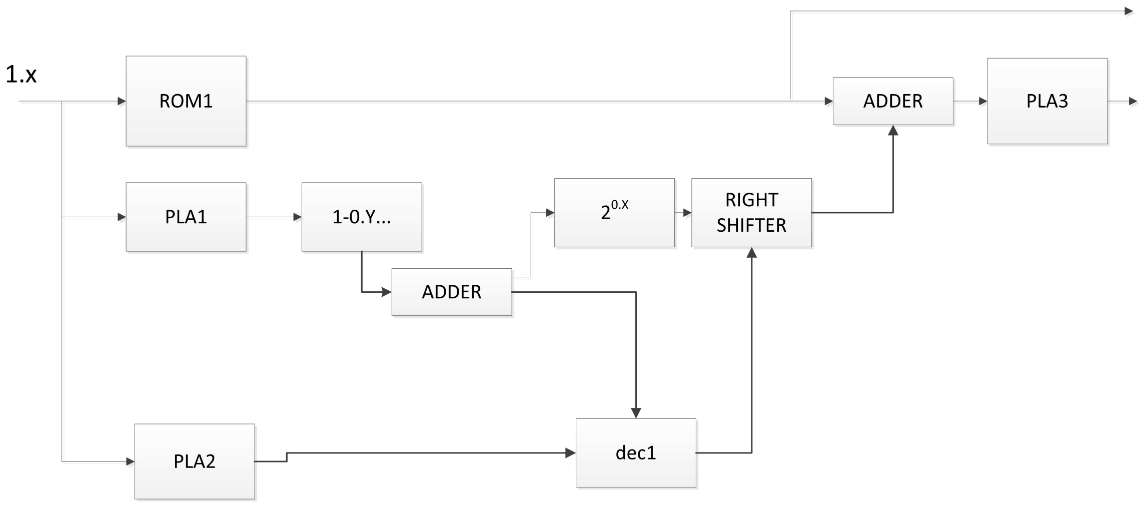


Figure 3.27: Huang et al. [97] Lin2Log architecture.

Stine and Schulte proposed approximation methods in [103–105] in 1997 and 1999 for functions like $log_2 x$, $2^x$, $ln(x)$, $x^{(1/2)}$ $and$ $sin(x)$. The two most commonly used configurations are the Symmetric Table addition Method (STAM) and Symmetric Bipartite Table Method (SBTM). Both configurations use Taylor first order approximation to the desired function avoiding use of a multiplier.

To approximate a function using bipartite tables, the input operand, $x$, defined by $n$ bits of resolution, is separated into three parts.

$$x = x_0 + x_1 + x_2 \tag{3.72}$$

With the length of each part defined by

$$n = n_0 + n_1 + n_2. \tag{3.73}$$

The Function is then approximated by the expression

$$f(x) \approx a_0(x_0, x_1) + a_1(x_0, x_2) \tag{3.74}$$

Here $n_0 + n_1$ are the most significant bits of $x$ are inputs to a table that provides the coefficients $a_0(x_0, x_1)$ and the $n_0$ most significant and $n_2$ least significant bits of $x$ are inputs to a table that contains the coefficients $a_1(x_0, x_2)$. The outputs from two tables are then summed to produce an approximation to $f(x)$ (see figure 3.28).

The technique can be extended by partitioning $x$ into $m+1$ parts $x_0, x_1, x_2, x_3.x_m$ with lengths of $n_0, n_1, n_2, n_3 n_m$ respectively.

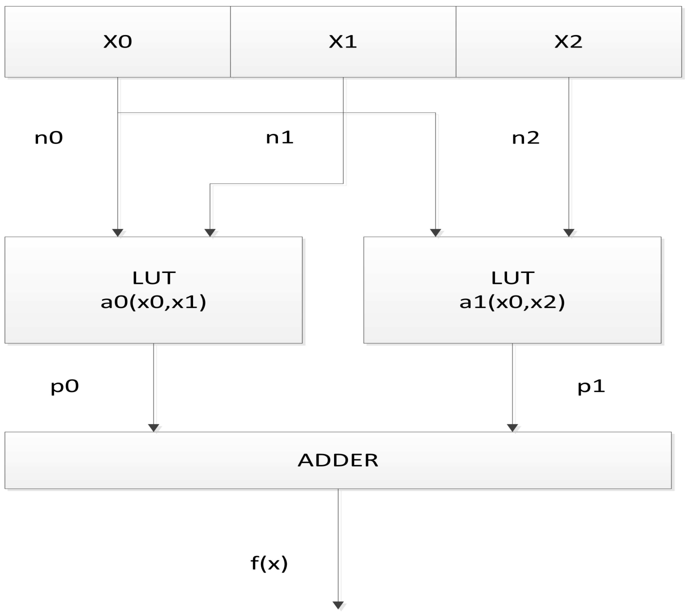


Figure 3.28: Stine and Schulte [103] Bipartite table method.

Larson [12] in 1994 published a new algorithm for accurately converting a floating point to a logarithmic number and accurately converting the result back to a floating point. Larson's algorithm [12] features the capability of performing floating point arithmetic functions in a single clock cycle with a high degree of accuracy for addition, subtraction, multiplication, division and square roots.

The algorithm in [12] firstly explains first order linear interpolation, in which floating point mantissa is converted to a logarithmic number fraction. The upper look up bits of the mantissa are used as an address into the look up table. In the look up table, each address is mapped to a corresponding SEED (original logarithmic number) value, which is the exact value of a logarithmic fraction for the high bits value of mantissa. Additionally, the address maps to a SLOPE value, which is the difference between the current SEED and previous value of SEED. Secondly an interpolator computes the logarithmic number value corresponding to the low order bits of the mantissa using a multiplier. The multiplier multiplies the low order bits by the SLOPE in order to interpolate the lower value of logarithmic function. Finally the SEED value and interpolated value are combined by using an adder to form a logarithmic number fraction of a given floating point number as shown in figure 3.29.

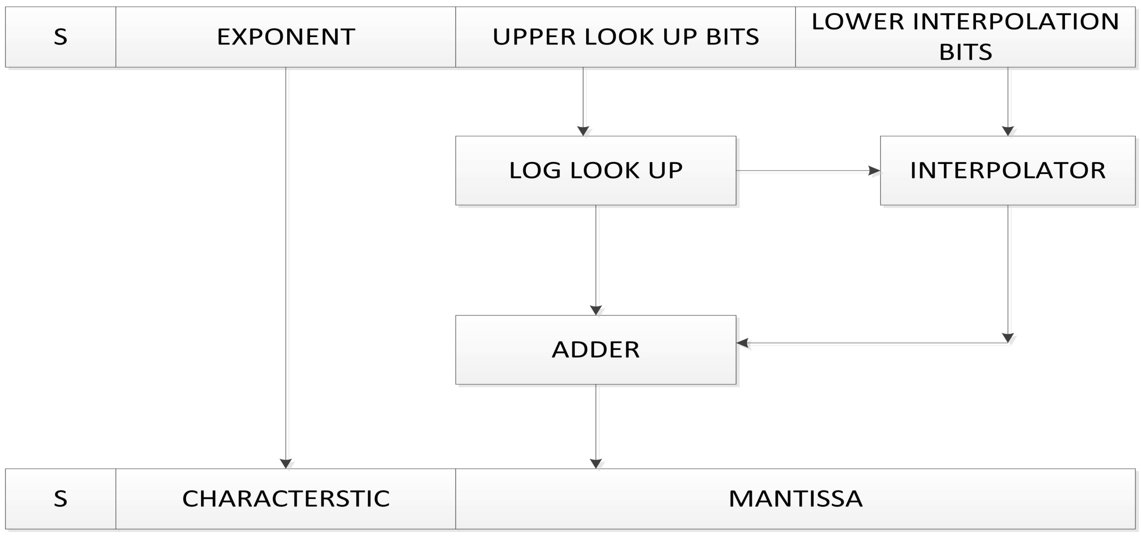


Figure 3.29: LARSON [12] first order interpolation.

In the second half of [12], Larson describes a second order interpolation method in order to improve the precision of the conversion. The second order interpolation method will reduce the look up table size to one quarter of the size needed for a first order interpolation look up table implementation. The floating point input number mantissa is divided into four parts: first order look up bits, $16-22$ bits, second order look up bits, $9-15$ bits, first order interpolation bits, $0-15$ bits, second order interpolation bits, $0-8$ bits. The first order look up bits are used as an address into a first order look up table. The first order interpolation bits are multiplied by the slope that is interpolated linearly to compute the corresponding logarithmic number value. The second order look up bits are used as an address into a second order look up table. The output from the second order look up table is a SEED. The SEED is added to an interpolated value derived from the second order interpolation bits to better interpolate the true logarithmic function and achieve greater accuracy. Logarithmic number values from all above steps are added to determine the resulting logarithmic number fraction value as shown in figure 3.30.

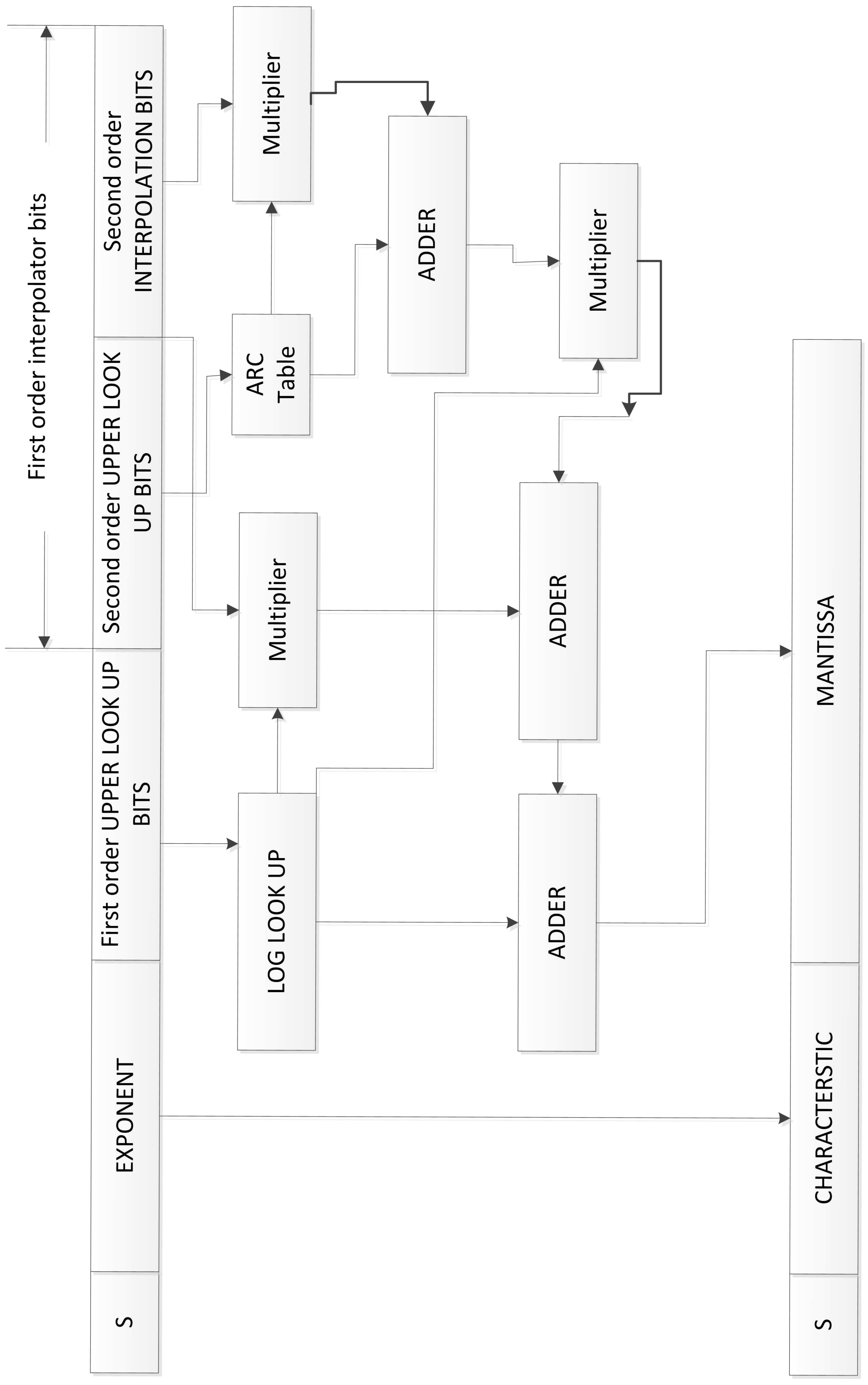

First order interpolator bits
Second order INTERPOLATION BITS
Second order UPPER LOOK UP BITS
First order UPPER LOOK UP BITS
EXPONENT
S
Multiplier
ARC Table
Multiplier
LOG LOOK UP
ADDER
Multiplier
ADDER
ADDER
MANTISSA
CHARACTERSTIC
S

Figure 3.30: LARSON [12] second order interpolation.

## 3.4 Summary

This chapter has provided a number of existing algorithms mentioned in patents, papers and electronic letters about Lin2Log and Log2Lin conversion methods proposed from the beginning to the recent approximations. The MATLAB simulation graphs provided in this chapter are used to understand different algorithms with the difference in their architecture used and accuracy achieved. In this chapter different algorithm approximation methods denote a trade between achievable accuracy, overall hardware costs and the speed of operations. However with time there has been a decrease in semiconductor device size and an increase in processing speed as stated by Moore's law in 1965 [106].

While performing a literature review on different algorithms, it was found that most of the logarithmic approximation architectures were not given in full detail, some of the papers/patents were just proposing algorithms without implementing them onto hardware. In some papers/patents the algorithms proposed were not simulated or their simulated information was not provided, which could help to study them in depth. The level of accuracy in few methods is also missing from some of the papers and especially patents. With the simulated MATLAB graphs provided in this chapter their accuracy achieved and hardware resources requirements are very much clearer.

In the past piecewise linear approximation could achieve higher level of accuracy but the problem was with the use of more intensive hardware, because of which non-linear piecewise approximations and solutions like Larson's algorithm [12] came into practical adaption as they reduced the hardware cost. However, today with FPGA devices (such as Xilinx Spartan and Virtex family) a large amount of memory is provided. Avoiding multiplication by approximation using the Taylor first order approximation is not needed any more as a large number of multipliers are provided on the FPGA itself.

In this chapter other algorithms mentioned for higher accuracy either require a vast amount of memory or complex architectures to perform logarithmic approximation, leading to intense usage of available hardware resource on an FPGA board. The aim of this research was to provide a simple (i.e. not complex) logarithmic converter without using too many hardware resources. The logarithmic converter is also required to perform complex arithmetic calculations in the logarithmic domain with high accuracy and due to these reasons, Larson's algorithm [12] was chosen for this research. The Larson algorithm [12] is not dependent on any further approximation methods such as the Taylor series, CORDIC etc., making it simpler to be implemented on an FPGA device. In [12] Larson does not describe the accuracy achieved by it when performing the second order interpolation. Furthermore, there is no information about the size of coefficient bits stored in the LUT. The information about the hardware implementation is missing from the Patent. The Larson algorithm [12] which is rarely referenced, leaves a lot of information to be discovered. The Larson algorithm [12] can be further improved and implemented in hardware for the same or higher accuracy up to 32 bits with hardware optimisation. This algorithm is further discussed in detail in Chapter 4.

# Chapter 4

# Improved Logarithmic Converter

## 4.1 Introduction

This chapter presents a novel PWL approximation-based Linear to Log (Lin2Log) converter [10]. The novel algorithm [10] is designed to perform the conversion of the decimal part of fixed/floating point input numbers to 23 bits of logarithmic fractional numbers. This chapter compares the performance of the proposed architecture with existing Lin2Log architectures based on traditional piecewise linear (PWL) [59, 61, 69, 70], piecewise polynomial (PWP) [72, 73, 91] and non-uniform piecewise [75, 95] approximations that have been presented in recent papers [107]. The proposed novel algorithm [10] is also found to have some similarities with algorithm mentioned in [108] describing a 2-step technique based on normalised difference functions.

This chapter assesses the performance of Larson algorithm [12] when used to convert normalized binary numbers with up to 23 fractional bits of accuracy. In Section 4.2 of this chapter the origins and characteristics of the Larson algorithm are described in detail. Section 4.3 proposes a novel algorithm including an analysis of its theoretical performance using MATLAB. Improvements to the basic Larson architecture are introduced. Section 4.4 details how the improved architecture has been implemented on an FPGA and Section 4.5 presents numerical data on the precision achieved and compares the FPGA resource utilisation with some recently published solutions.

## 4.2 Larson Lin2Log conversion algorithm

A binary logarithm of a number $x$ is typically defined using the 4-tuple $(Z, S, I, F)$

$$x = 2^{log_2 x} = (1 - Z).(-1)^S.2^I.2^{0.F} \tag{4.1}$$

where $S$ is the sign bit, $I$ the integer part and $F$ the fractional part (or mantissa) of the logarithm base 2 respectively and $Z$ is used to represent the special case of $x = 0$. The derivation of $Z$, $S$ and $I$ is reasonably straightforward and examples can be found in [54]. The conversion of a fixed/floating point number to and from the log domain requires the approximation of the non-linear terms $Log_2(1.F)$ for Lin2Log and $2^{0.F}$ for Log2Lin. In this chapter we consider that the input fixed/floating point number is normalised using LOD or LZD techniques (see sections 3.2.1 and 3.2.2). So, the input fixed/floating point number becomes $1 \leq 1.F < 2$, where $F$ is the fractional component of a normalised number. Many algorithms for approximating $log_2 1.F$ have been based on improvements to the simple linear interpolation approximation first proposed by Mitchell [59] in 1962. Although this approximation required minimal hardware, Mitchell [59] showed that the conversion error, e, of this interpolation is in the range $0 \leq e \leq 0.08639$ which is equivalent to about 4 bits of binary precision (see section 3.3.1).

Subsequent papers have proposed improvements to the basic Mitchell [59] architecture which have been achieved through the use of more complex curve-fitting techniques and arithmetic components (adders and shifters) without requiring LUT or complex arithmetic elements i.e. hardware multipliers [61, 91, 109]. However as technology has improved and the demands for higher accuracy have increased there has been a greater reliance on LUT techniques based on uniform PWL, PWP approximations [96, 110, 111]whereby the size of the LUTs, which become prohibitive for resolutions greater than 16 bits, have been reduced at the cost of additional arithmetic components  adders and multipliers. Most recent algorithmic improvements have been based on novel methods for reducing the size of the LUT and the complexity and number of arithmetic components used [112]. Not all of these have appeared in the academic literature and some are rarely referenced or used for performance comparison [113–117].

In [75] architectures for achieving double-precision FP (Floating Point) accuracies have been reported where non-uniform polynomial techniques have been used to keep the size of the LUTs within acceptable bounds, albeit at the cost of more complex address encoding (see section 3.3.3).

Many early Lin2Log solutions focused on implementation using custom logic in VLSI technology but advances in modern FPGA device densities have increased interest in the use of logarithmic techniques for signal processing on FPGAs using the hardwired arithmetic and memory resources available in most modern FPGA fabrics [118]. For floating point and higher accuracy the substantial amounts of memory are still required to implement such converters even when using piecewise techniques and this remains a limiting factor, even on modern FPGAs. A further limitation when implementing these converters on FPGAs is that the embedded processing and memory blocks come in discrete sizes. In Xilinx devices the minimum size of Block-RAM (BRAM) memory is 18 kbits and hardwired multipliers have a fixed size of $18 \times 18$ (or $25 \times 18$) bits. Although on some FPGA devices it is now possible to partition the BRAM into smaller blocks or use distributed memory elements in the configurable fabric the algorithms often do not make efficient use of BRAM elements. Although increasing with every new generation of FPGA devices, the number of BRAM elements available remains a limited resource and where possible it is important for conversion algorithms to optimize their use.

The Larson algorithm [12] (discussed in section 3.3.5) combines two PWL approximations to convert a normalised binary input,$1 \leq x < 2$ , with 23 bits of fractional precision into a binary logarithm $0 \leq log_2(x) < 1$ also with 23-bits of fractional precision. The first PWL stage is used to approximate the Log2 curve while the second is used to reduce the residual conversion error, produced by the first PWL approximation, to an acceptable level. The improvement to the basic PWL approximation proposed by Larson has some similarities with the algorithms first proposed by Marino [72], Combet [60], Kmetz [69] and Maenner [70] (see section 3.3.2).

Larson [12] proposed an improvement to the Maenner algorithm to produce an approximation to the binary logarithm, $alog_2(1.F)$, that is

$$alog_2(1.F) \simeq log_2(1.F) \tag{4.2}$$

where

$$F = F_{k-1} \times 2^{-1} + F_{k-2} \times 2^{-2} + ...... + F_0 \times 2^{-k} = \sum_{i=1}^{k} F_{k-i} 2^{-i} \tag{4.3}$$

Larson's proposal [12] is based on a classic PWL architecture and began with an evaluation of the accuracy achievable using such an architecture over a range of resolutions. The table presented by Larson is reproduced here for clarity in table 4.1, which shows the accuracy achieved using a first order (PWL) as presented in [12]. For instance this table shows that achieve an accuracy of 14 bits is achieved using a look up address width of 6 bits and 9 interpolating data bits. In [12], the size of the coefficient bits stored in the LUT used to achieve the listed accuracy is missing. The information about the required accuracy achieved when using the two stage Larson algorithm is set for future work. Table 4.1 also shows that the number of LUT interpolation data bits used for a single PWL approximation is 3 more than the number of address bits used for the zeroth order LUT. Larson's scheme aimed at reducing the size of the interpolation LUT, which has a significant impact on the total memory required for the conversion algorithm.

Table 4.1: Larson table of maximum errors using PWL Approximation [12]

| Look-up bit size | Interpolate bit size | Maximum error | Bits of accuracy |
|---|---|---|---|
| 6 | 9 | $4.335 \times 10^{-5}$ | 14.49 |
| 7 | 10 | $1.092 \times 10^{-5}$ | 16.48 |
| 8 | 11 | $2.741 \times 10^{-6}$ | 18.48 |
| 9 | 12 | $6.866 \times 10^{-7}$ | 20.47 |
| 10 | 13 | $1.718 \times 10^{-7}$ | 22.47 |
| 11 | 14 | $4.297 \times 10^{-8}$ | 24.47 |
| 12 | 15 | $1.075 \times 10^{-8}$ | 26.47 |
| 13 | 16 | $2.687 \times 10^{-9}$ | 28.47 |
| 14 | 17 | $6.718 \times 10^{-10}$ | 30.47 |
| 15 | 18 | $1.679 \times 10^{-10}$ | 32.47 |
| 16 | 19 | $4.199 \times 10^{-11}$ | 34.47 |

The basic architecture proposed by Larson is shown in figure 4.1. The first stage uses a piecewise linear approximation between a set of exact points.

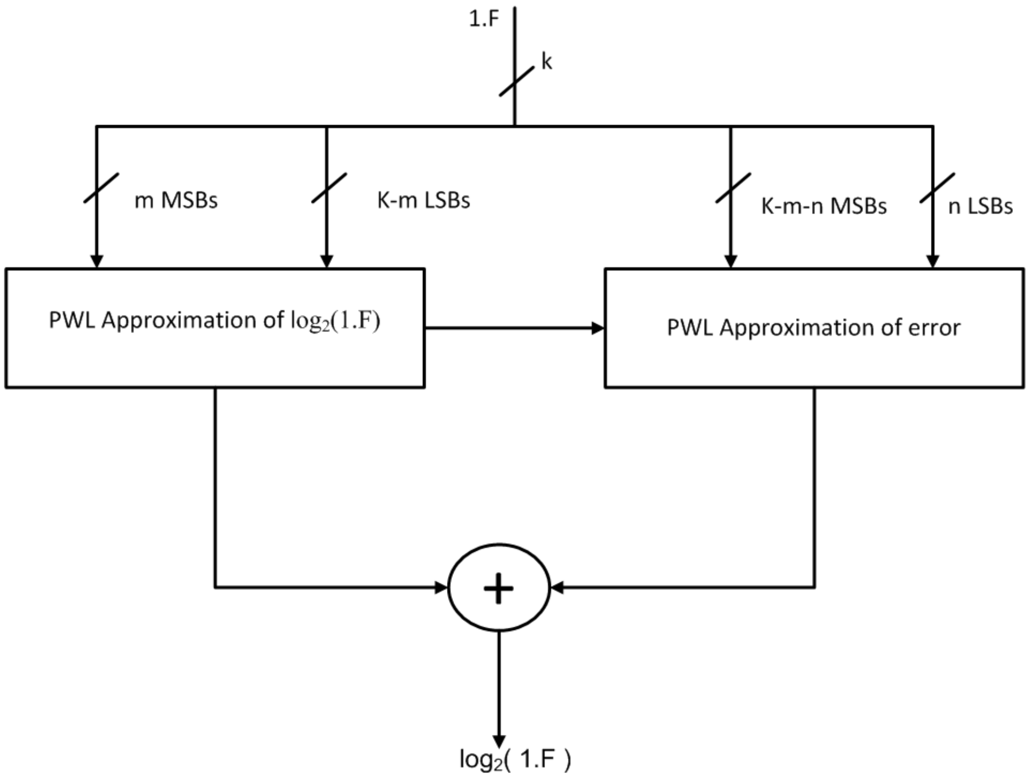


Figure 4.1: Larson algorithm

F is partitioned into $p = 2^m$ segments using the $m$ MSBs (Most Significant Bits) of F. Each segment contains $2^n$ elements, where $m = k - n$. For each of the $p = 2^m$ segments a unique pair of PWL coefficients $a_p$ and $b_p$ are stored in the LUT and $alog_2(1.F)$ is calculated using (4.3)

$$alog_2(1.F) = a_p + b_p(\sum_{i=k-m-1}^{k} F_{k-i}2^{-i}). \tag{4.4}$$

In Kmetz an LUT is used to approximate the difference (or error), $\varepsilon$ , between $alog_2(1.F)$ and $log_2(1.F)$, that is

$$\varepsilon = log_2(1.F) - alog_2(1.F). \tag{4.5}$$

where the n least significant bits of $1.F$ are used as the address of the LUT containing $2^n$ values of $\varepsilon$. Hence giving

$$alog_2(1.F) = a_p + b_p(\sum_{i=k-m-1}^{k} F_{k-i}2^{-i} + \varepsilon\ ). \tag{4.6}$$

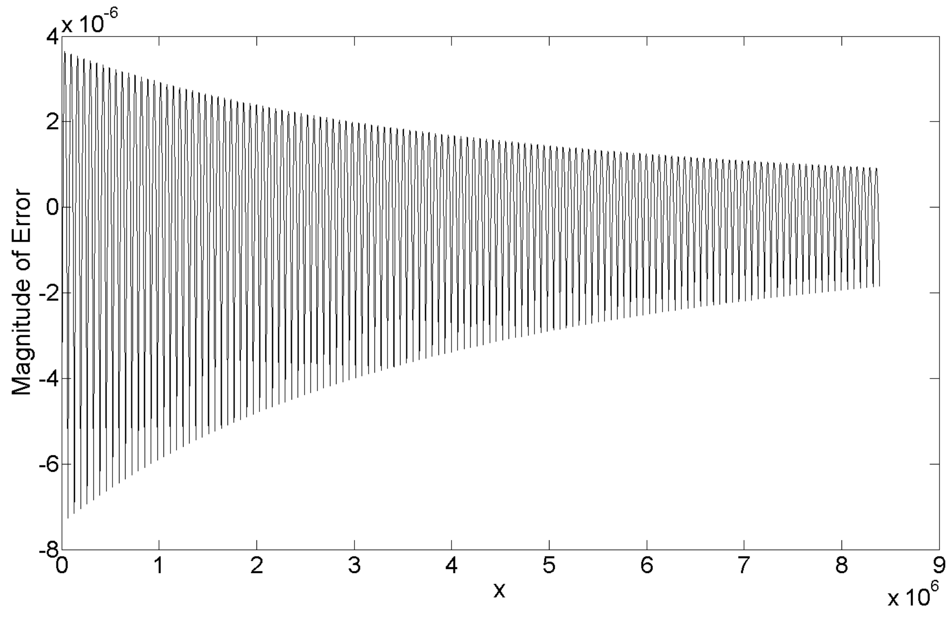
by changing this to

$$alog_2(1.F) = a_p + b_p(\sum_{i=k-m-1}^{k} F_{k-i}2^{-i} + S_p\varepsilon'). \tag{4.7}$$

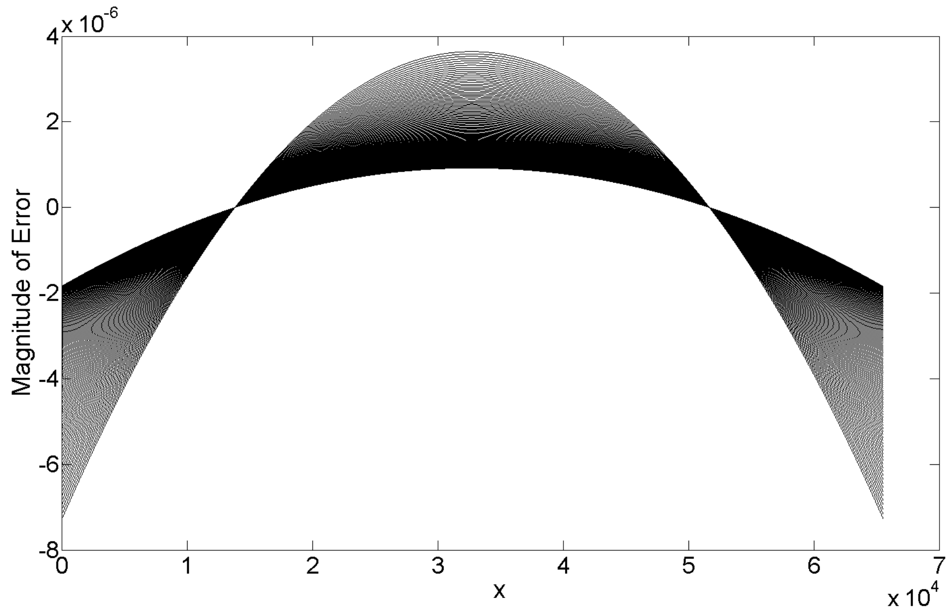
where $S_p$ is a unique factor in each segment used to scale a PWL approximation of the error, $\varepsilon'$, in each of the p segments, that is

$$\varepsilon' = a'_j + b'_j(\sum_{i=k-m-1}^{k} F_{k-i}2^{-i}) \quad j = 1 : n. \tag{4.8}$$

Larson [12] noticed the similarity in the shape of the error curves produced between secant points of the PWL approximation. Figure 4.2a shows an example of the curves produced when the log conversion curve for a normalised input with 23 fractional bits of precision is approximated using a PWL approximation with $m = 7$ (128 segments) with each segment containing $2^{16}$ points. Larson proposed using a PWL approximation to a single error curve with the zeroth and first order coefficients ($a'$ and $b'$) stored in an additional LUT (LUT2). The resulting error curve is scaled using a unique multiplicative factor ($s$) for each pair of PWL coefficients ($a$ and $b$) stored in LUT1 for the initial PWL approximation of $log_2(x)$. The error curve used in each segment is a scaled version of the composite error curve derived from the error curves produced in first stage. A MATLAB simulation of first stage error curves is shown in figure 4.2a. The first stage error curves (shown in 4.2a) are of similar shapes but with different scaling factors (due to same uniform step size) and are shown in figure 4.2b. The mean and normalisation of these error curves is shown in figure 4.3.

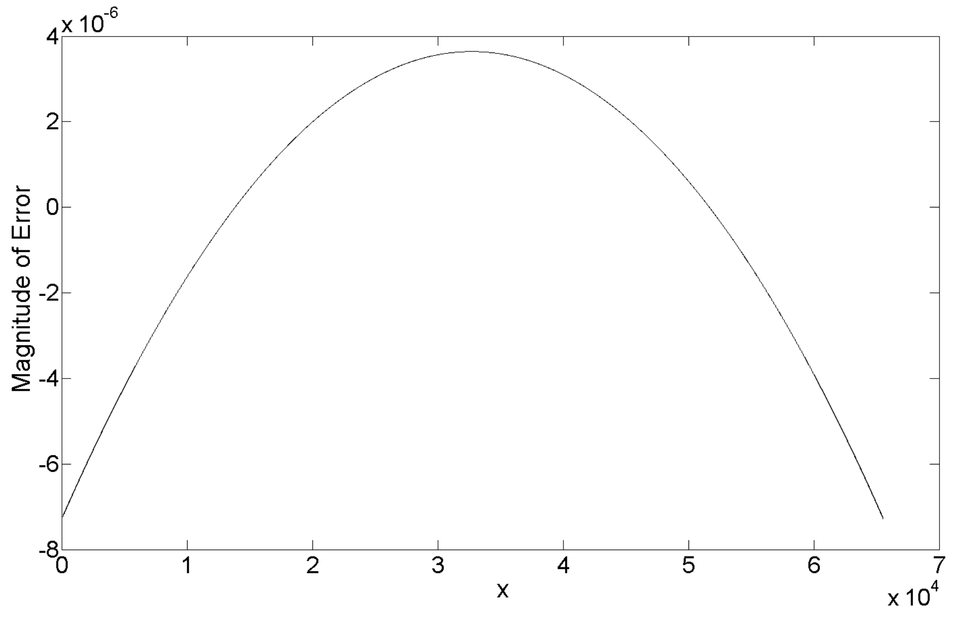


(a)

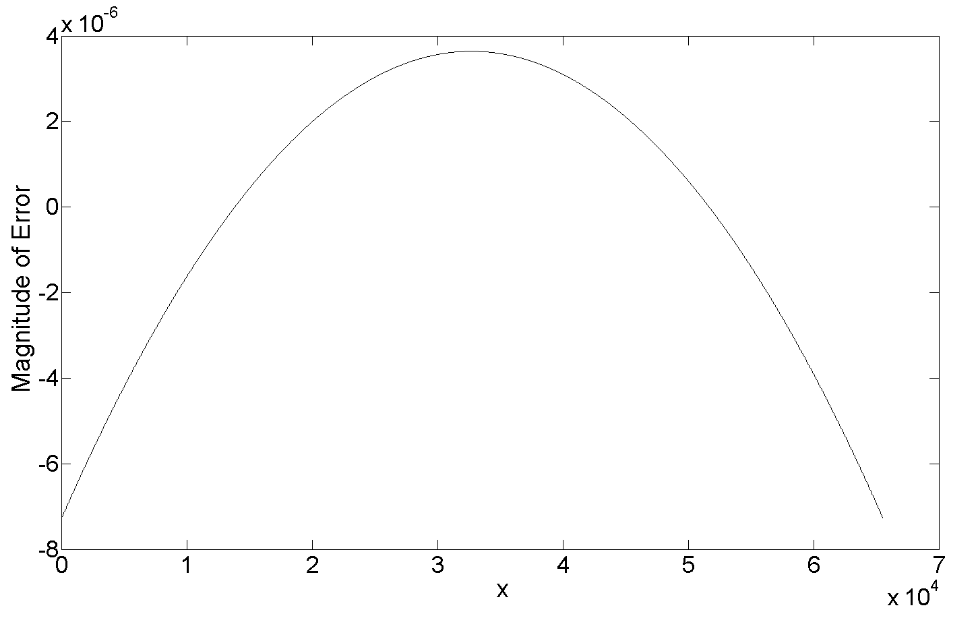


(b)

Figure 4.2: Larson first stage errors (128 segments, linear Interpolation) (a) Error for 128 intervals (b) Superimposed error curves for each segment.



(a)

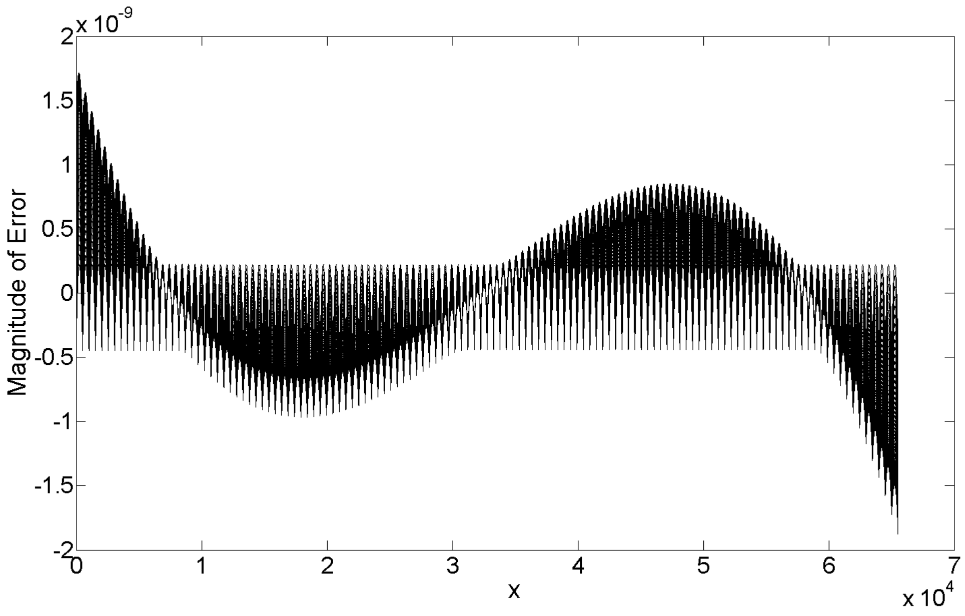


(b)

Figure 4.3: Larson first stage errors (128 segments, linear Interpolation) (a) Normalised error curves for each segment (b) mean of normalised curves.

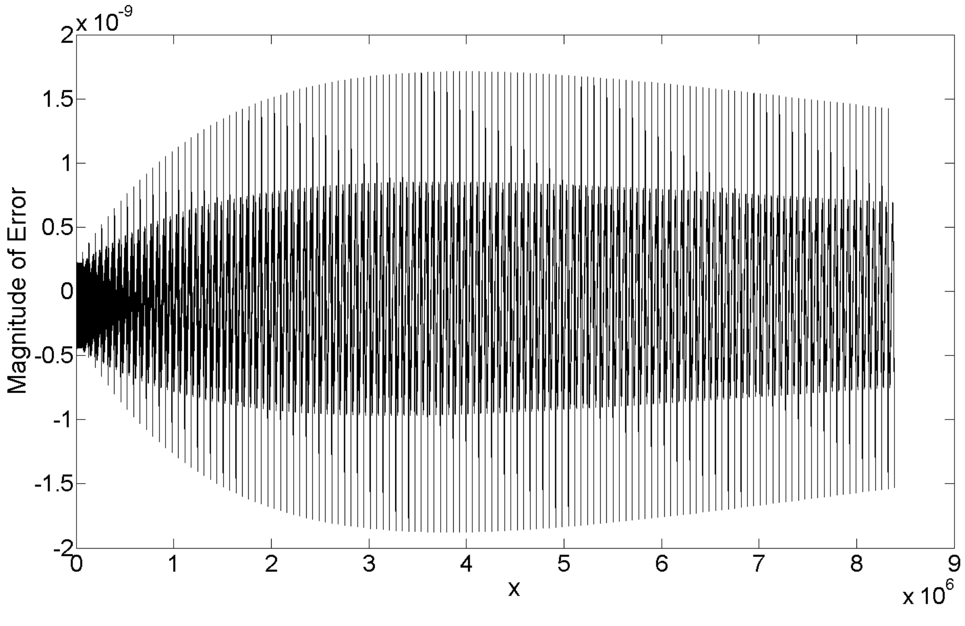
## 4.3 Novel Algorithm

The novel algorithm [10] proposed in this chapter comprises two stages where the first stage is similar to the algorithm proposed in [12]. However, in the second stage the novel algorithm [10] evaluates the dependency of the error curve by using three possible composite error curves. These three possible composite error curves in this chapter have been evaluated using MATLAB. The errors produced when using a composite error curve based on the scaled Mean, Max or Min of the error curves are shown in figures 4.4 and 4.5. The version shown in figures 4.4a and 4.4b shows the smallest overall error and the RMS (Root Mean Square) of the respective residual error when the composite error is approximated using a scaled linear approximation in the second stage. Figures 4.4 and 4.5 show clearly how the distribution of the error changes when the composite curve is derived using different criteria (min, max and mean). The MATLAB simulations show that when the max error curve is used as a stored error curve in LUT for second stage approximation, it minimises the overall RMS error value over the range $1 \leq log_2(1.x) < 2$ (shown in figure 4.4b). Hence the max error curve for second stage approximation is the most appropriate choice.



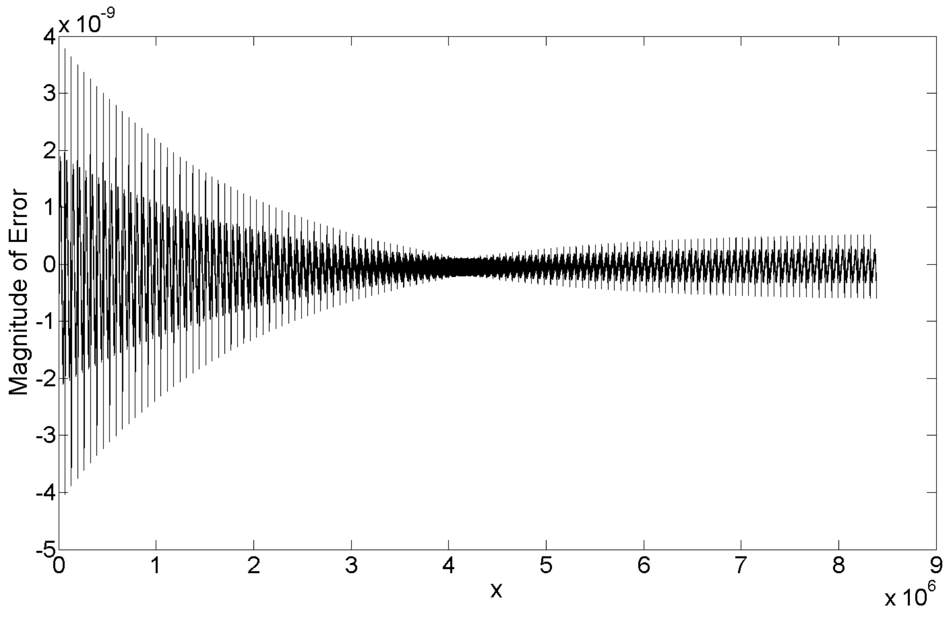
(a)

Figure 4.4: Larson total error i.e. (1st + 2nd ) stage errors (a) Superimposed error curves for each segment errors using maximum error curve (i.e. 1st error curve) of first stage



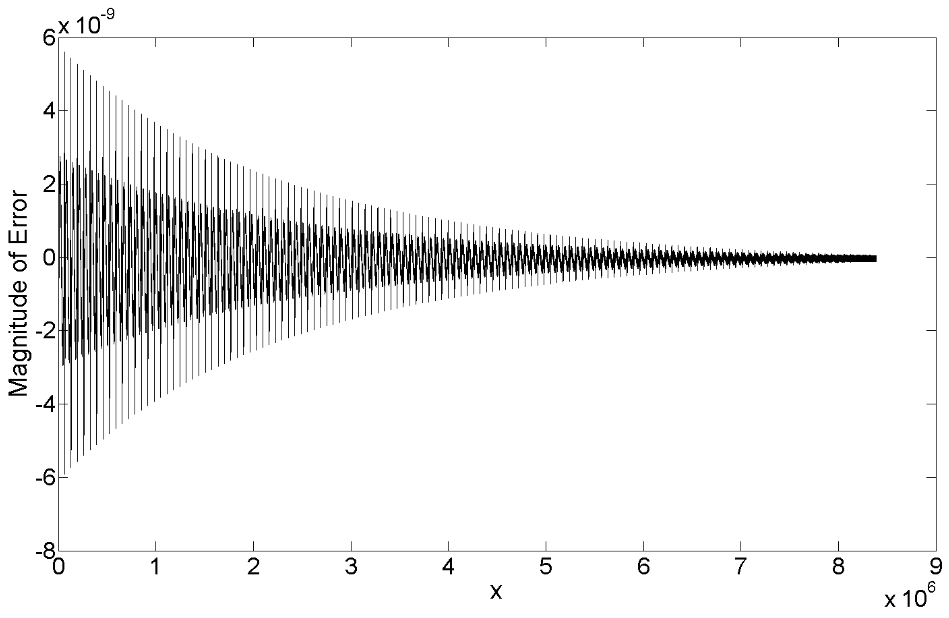
(b)

Figure 4.4: Larson total error i.e. (1st + 2nd ) stage errors (b) error obtained using maximum error curve (i.e. 1st error curve) of first stage.



(c)

Figure 4.5: Larson total error i.e. (1st + 2nd ) stage errors (a) error obtained using mean (average i.e. 64th error curve) error curve of first stage



(d)

Figure 4.5: Larson total error i.e. (1st + 2nd ) stage errors (b) error obtained using minimum (i.e. 128th error curve) error curve of first stage.

So far the effects of finite precision in the LUT data and the multipliers have been ignored but need to be considered as they will impact on the effectiveness of the hardware implementation. The proposed algorithm was tested using a number of different data partitions for a normalised binary number with 23 bit fractional bits. These are summarised in table 4.2 and their respective MATLAB simulation plots are shown in appendix A. In this table a configuration of 7:16::7:9 indicates that 7 the first MSBs, are used as the address of a LUT1 and 16 LSBs are used for the calculation of the first order term of the PWL approximation. The next 7 significant bits (bits 15 down to 9) are used as the address for the second LUT and the remaining 9 bits of the input (bits 8 down to 0) are used for the calculation of the first order term of the correction. The different data partitions tested are defined in a similar way.

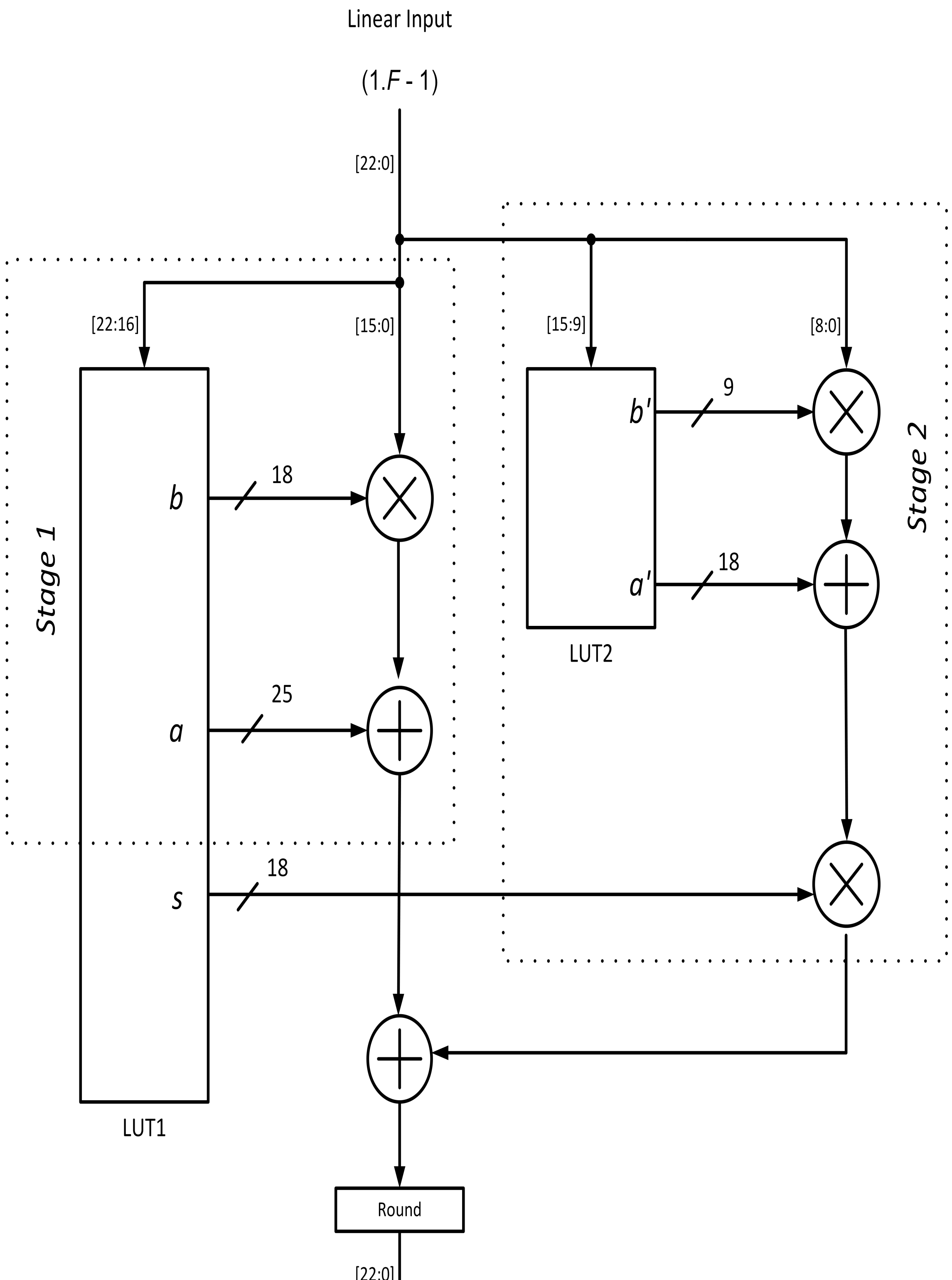
Table 4.2: Ideal Performance using Scaled Max Error Curve for 23 bit fractional input (MATLAB simulation results)

| Configuration | LUT1 | | Input Bits | LUT2 | | RMS ($\times 10^{-9}$) | Accuracy Bits |
|---|---|---|---|---|---|---|---|
| | Add Bits | Interpolation Bits | | Add Bits | Interpolation Bits | | |
| 7:16::7:9 | 7 | 16 | 16 | 7 | 9 | 1.796 | 29 |
| 7:16::8:8 | 7 | 16 | 16 | 8 | 8 | 1.699 | 29 |
| 8:15::6:9 | 8 | 15 | 15 | 6 | 9 | 0.401 | 31 |
| 8:15::7:8 | 8 | 15 | 15 | 7 | 8 | 0.246 | 31 |
| 8:15::8:7 | 8 | 15 | 15 | 8 | 7 | 0.218 | 32 |

The effects of finite precision coefficients in LUT1 and LUT2 were considered for each of the above configurations and the accuracy obtained is summarised in table 4.3. Their MATLAB simulation plots are shown in appendix B. It has been assumed that the default size of the multiplier inputs is a maximum of 18 bits as is the case with most Xilinx FPGA devices. The final output of approximation is rounded to 23 bits using a rounding to nearest algorithm. A detailed view of algorithm architecture with finite precision coefficients in LUT, multipliers and adders size are is shown in figure 4.6.

Table 4.3: LUT coefficient bits (MATLAB simulation results)

| Config | LUT1 | | | | LUT2 | | | Total Bits | RMS |
|---|---|---|---|---|---|---|---|---|---|
| uration | a | b | s | LUT1 Bits | a' | b' | LUT2 Bits | | ($\times 10^{-7}$) |
| 7:16::7:9 | 25 | 18 | 18 | 7808 | 18 | 9 | 3456 | 11264 | 1.03 |
| 8:15::6:9 | 25 | 18 | 18 | 15616 | 18 | 9 | 1728 | 17344 | 0.8 |
| 8:15::7:8 | 25 | 18 | 18 | 15616 | 18 | 9 | 3456 | 19072 | 0.8 |
| 8:15::8:7 | 25 | 18 | 18 | 15616 | 18 | 9 | 6912 | 22528 | 0.8 |



## 4.4 Implementation

The architecture circuit shown in figure 4.6 and has been implemented onto a Spartan3 (XC3S200) and a Spartan6 (XC6SLX16) FPGA device using Xilinx ISE V13.2 synthesis tools and a Xilinx CORE generator to compare the effect of differences in the configurable fabric on the efficiency of the implementation. Synthesis was performed for both optimised area and speed. To improve speed additional pipelining stages have been added to the arithmetic stages. The post-implementation results are summarised in table 4.4, where maximum and minimum pipelining are shown with respective to their frequencies. The FPGA design top level diagram for maximum and minimum pipelining of novel algorithm [10] architectures are shown in figures 4.7 and 4.8.

Table 4.4: Implementation statistics

| Spartan3 utilisation | BRAM | HMUL | Logic slices | speed,MHz | Latency,clks |
|---|---|---|---|---|---|
| min pipelining | 2 | 3 | 104 | 42.3 | 2 |
| max pipelining | 2 | 3 | 232 | 127.8 | 8 |
| mult 3 implemented using CLB | 2 | 2 | 286 | 128 | 8 |
| **Spartan6 utilisation** | **BRAM** | **DSP48** | **Logic slices** | **speed,MHz** | **Latency,clks** |
| min pipelining | 2 | 3 | 28 | 71.4 | 2 |
| max pipelining | 2 | 3 | 55 | 160 | 8 |
| mult 3 implemented using CLB | 2 | 2 | 83 | 180 | 8 |

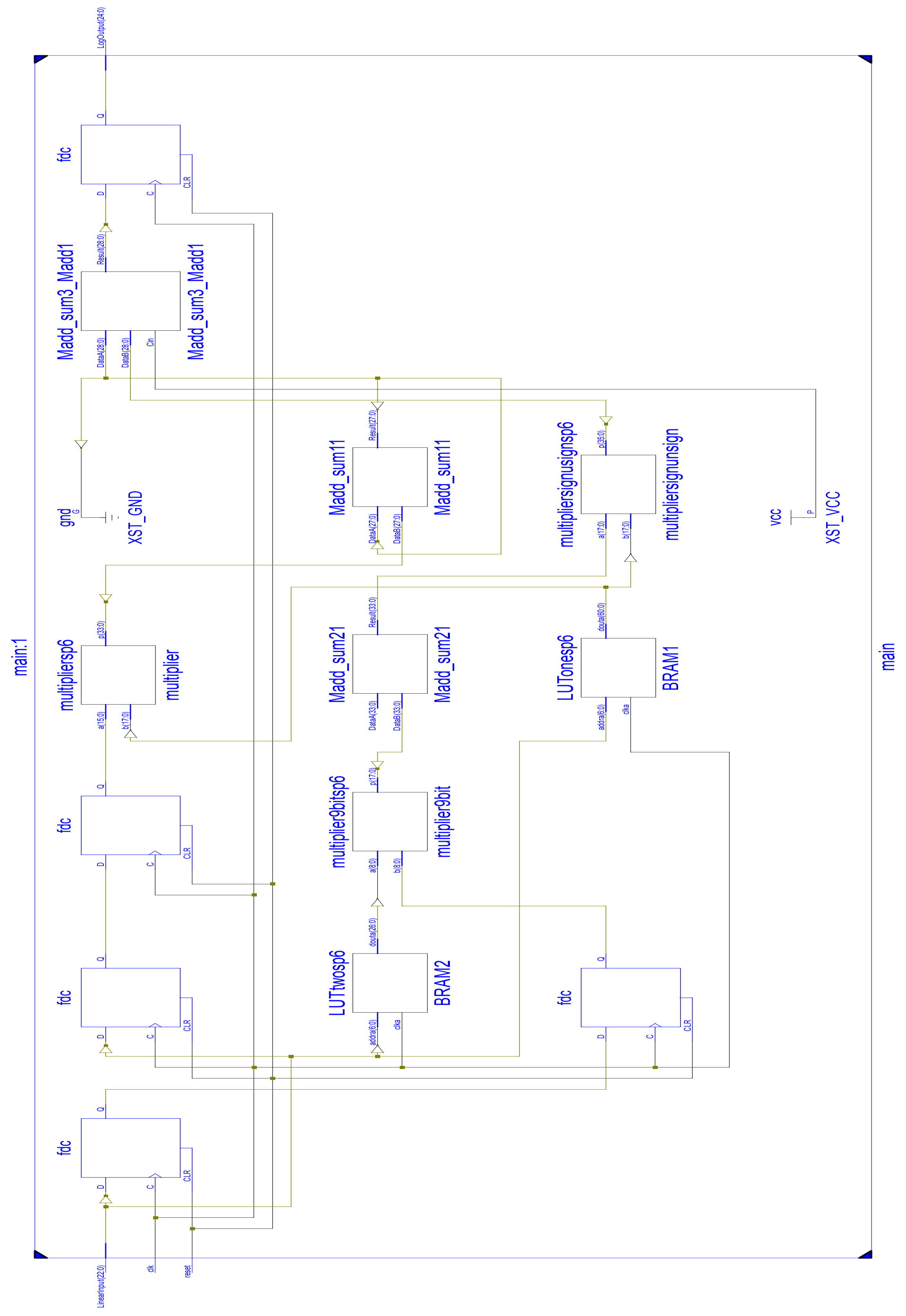


Figure 4.7: Spartan 6 based minimum pipelining design of the novel algorithm

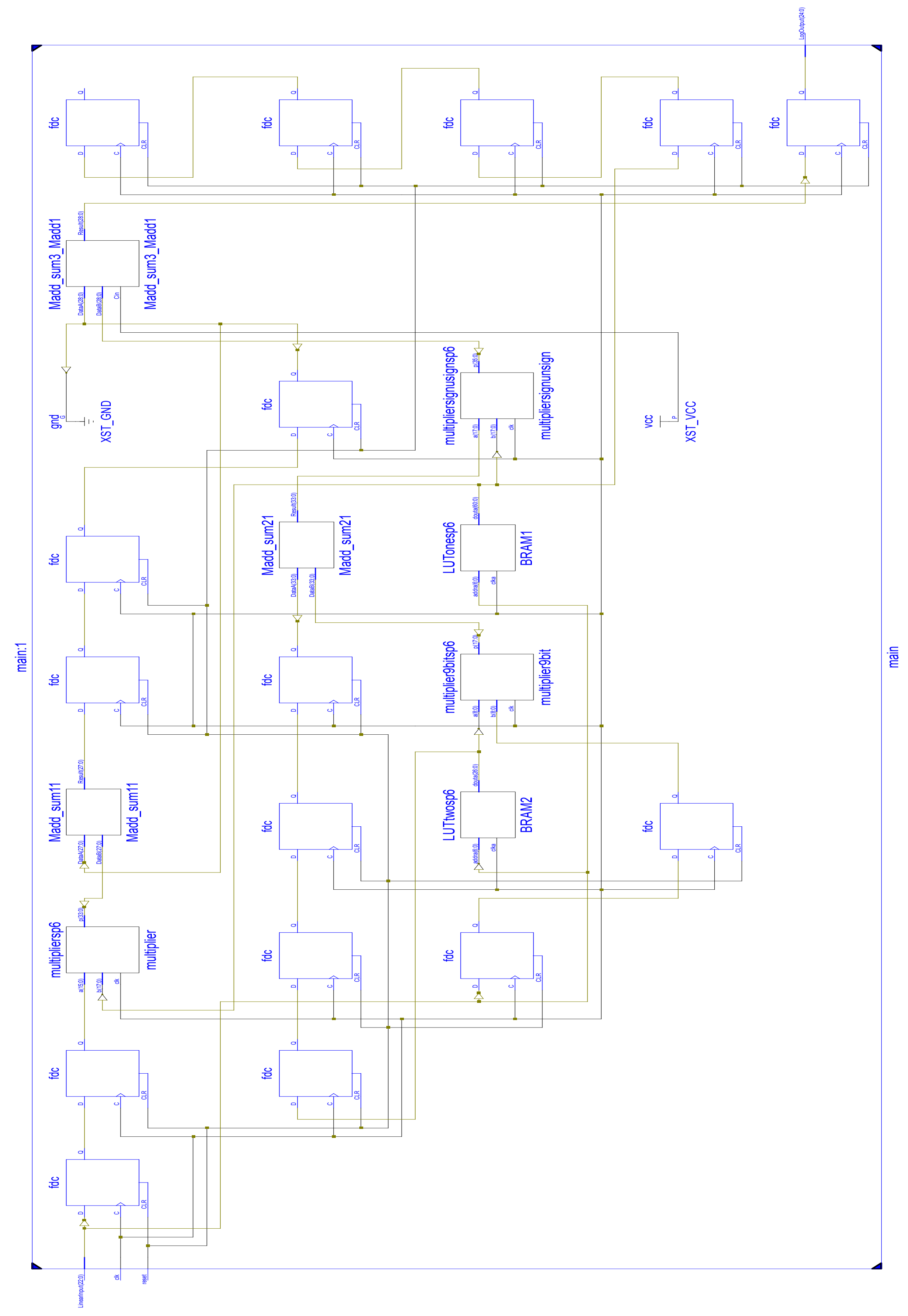


Figure 4.8: Spartan 6 based maximum pipelining design of the novel algorithm

## 4.5 Results

The results in table 4.3 indicate that the proposed algorithm [10] when built using a second stage approximation achieves accuracies equal to or better than recent solutions proposed in [119], [120] and [118] that used piece-wise polynomial or non-uniform piecewise polynomial techniques for Lin2Log conversion with 23 bits of fractional precision while using less LUT resources. A comparison with these techniques is shown in table 4.5, where $F$ stands for faithful rounding units and $B$ stands for BTFP (Better than Floating Point) units as described in [118].

Table 4.5: Comparison of Results with previous work for Single Floating Point Precision.

| Source | Device | BRAM 18 k | BRAM 9 k | HMUL/DSP48 | slices |
|---|---|---|---|---|---|
| [120] | Altera Stratix | - | - | 2 | 234 |
| [119] | Virtex II XC2V2000 | 2 | 0 | 3 | 286 |
| [118]$F$ | Virtex II XC2V6000 | 2 | 0 | 3 | 286 |
| [118]$B$ | Virtex II XC2V6000 | 2 | 0 | 3 | 327 |
| Table 4.4 | Spartan 3 XC3S200 | 2 | 0 | 3 | 232 |
| Table 4.4 | Spartan 6 XC6SLX16 | 1 | 1 | 3 | 55 |

NFGs cannot be mapped into the FPGA because of the excessive memory size.

This chapter compares logic utilisation of the proposed algorithm with [118] and [119] using the Xilinx Virtex-II family. Although a different device, the underlying architecture of the slices is the same. Because of this there is almost no difference in the number of logic slices required by both the Spartan3 and the Virtex II devices.

The only significant change is the substantial increase in the operating frequency of the algorithm on a Virtex II device due to the fabrication technology used (not the architecture).

In the proposed algorithm, with the same number of hardware multipliers and BRAM used in [118], a reduction of 18.88% to 29.05% of the required FPGA slices is achieved using a Spartan3 device. This reduction in FPGA slices is further improved with the improved slice architecture available on a Spartan6 device.

In Spartan6 BRAM memory blocks can also be more efficiently partitioned than in Spartan3 devices. Hence in table 4.5 the Spartan6 solution uses only one block of 18k BRAM and one block of 9k BRAM. The NFG (Numerical Function Generator) presented in [120] can be used for a variety of functions without introducing any modifications, but due to this flexibility its LUT uses memory sizes up to 5,586,944 bits for the $ln(x)$function. This is clearly larger than the algorithm proposed in this chapter and will not map onto an FPGA's memory resources.

The algorithm presented in [121] shows that the memory used in the 2nd order polynomial for 24-bit precision (7,324 bits) is lower than has been achieved in the 7:16 configurations in table 4.3. However, the architecture is much more computationally intensive compared to the algorithm presented here. The computational overhead also increases the calculation time which is 38.9 ns (approx. 25.7 MHZ) for 16-bit precision (implemented on Xilinx Virtex-II pro) in [121], whereas the log converter presented here has a worst case calculation time (i.e. no pipelining implemented) on a Spartan3 of 23.64 ns (approx. 42.3 MHZ ) for 23 bits of accuracy. In [122], the author presented a search algorithm for optimising a conversion function. For numerical functions over 2000 designs are considered. [122] concluded that for $ln(1 + x)$ a 3rd order polynomial solution was the most efficient. Although the results in [122]) are achieved using a smaller memory footprint but at a cost of substantial increase in the arithmetic components required, which offsets any advantage gained by this approach [122].

## 4.6 Conclusion

This chapter has demonstrated an efficient two stage PWL novel algorithm [10] for converting a fixed / floating point input normalised binary number (in the region $1 \leq 1.x < 2$) to an equivalent logarithmic number. The novel algorithm [10] in this chapter unlike in [12] uses all possible combinations for choosing the generic error curve to be used in its second stage approximation. This chapter shows an implementation of a two stage PWL algorithm on a legacy and a modern FPGA device. This chapter also provides quantitative data about the overall reduction in LUT memory achieved while using the proposed algorithm. The hardware implementation of the proposed algorithm shows that it requires fewer arithmetic components to achieve 23 bits of fractional precision than other algorithms using uniform and non-uniform piecewise linear or piecewise poynomial techniques and require less than 20 kbits of ROM and a maximum of three multipliers. The chapter presents empirical data for the accuracy of the conversion using a number of different LUT configurations. Example implementation statistics have been presented using Xilinx Spartan3 and Spartan6 device families. Synthesis results confirm that the algorithm operates at frequency of 42.3 MHz on a Spartan3 device and 127.8 MHz on a Spartan6 with a latency of two clocks. This increase to 71.4 and 160 MHz, respectively, when the latency is increased to eight clocks. On a Spartan6 XC6SLX16 device, the converter uses just 55 logic slices, three multipliers and 11.3kbits of Block RAM configured as ROM.

# Chapter 5

# Further Improved Logarithmic Converter

## 5.1 Introduction

This chapter presents a novel algorithm [11], performing further improvements on the algorithm [10] previously presented in section 4.3. The proposed novel algorithm [11] exploits the symmetrical properties of a normalised error curve stored in the second stage of the algorithm [10]. This chapter shows that by exploiting the symmetrical properties of the normalised error curve, the size of the second stage LUT is reduced significantly, allowing it to be implemented more efficiently using the combinatorial logic (CLB) available in the reconfigurable fabric instead of a second BRAM. The results of the optimisations of the proposed algorithm [11] on Xilinx Spartan3 and Spartan6 families are presented in this chapter and are compared with the previously proposed algorithms.

The chapter begins with describing a novel algorithm [11] in section 5.2, including its theoretical performance using MATLAB. Section 5.3 details how the novel architecture has been implemented on new and legacy FPGA devices. Results including numerical data on the precision achieved and comparison with some recent published solutions are presented in section 5.4.

## 5.2 Novel Algorithm

The algorithm presented in Chapter 4 is implemented with two stages i.e. stage 1 and stage 2 (previously mentioned in section 4.2) as shown in figure 4.6. Stage 1 contains a PWL approximation of the log function. The coefficient LUT contains values for $a_p$ and $b_p$ and also the scaling factor $S_p$ as described in equations 4.6 and 4.7. Stage 2 is a PWL approximation of the error curve and contains an additional multiplier to scale the curve. It should be noted that the error curve generated for LUT2 is using a second-stage PWL approximation. The scaled version of normalised error curve is then added to the first-stage PWL approximation of the log function to reduce the overall conversion error. LUT1 is used to store the zeroth (a) and first order (b) coefficients of the PWL approximation together with a scaling factor (s) which is used to multiply the normalised error curve approximated using PWL coefficients (a' and b') and stored in LUT2.

Section 4.2 showed some analysed and verified simple methods for deriving a curve that produces the minimum (root mean square) error approximation. This was assessed for a normalised input with 23 bits of fractional precision. Each configuration produced a conversion error of less than 1ULP (Unit of Last Place) where 1ULP $= 2^{-23} = 1.19 \times 10^{-7}$. The results presented in section 4.3 for different LUT sizes for a 23-bit conversion in the previously proposed algorithm [10] are presented in table 5.1.

Table 5.1: LUT coefficient bits

| Configuration | LUT1(a=25, b=18, s=18) | | | LUT2(a'=18, b'=9) | | | Total | RMS |
|---|---|---|---|---|---|---|---|---|
| | Add Bits | Data Bits | LUT1 Bits | Add Bits | Data Bits | LUT2 Bits | Bits | ($\times 10^{-7}$) |
| 7:16::7:9 | 22:16(7) | 15:0(16) | 7808 | 15:9(7) | 8:0(9) | 3456 | 11264 | 1.03 |
| 8:15::6:9 | 22:15(8) | 14:0(15) | 15616 | 14:9(6) | 8:0(9) | 1728 | 17344 | 0.8 |
| 8:15::7:8 | 22:15(8) | 14:0(15) | 15616 | 14:8(7) | 7:0(8) | 3456 | 19072 | 0.8 |
| 8:15::8:7 | 22:15(8) | 14:0(15) | 15616 | 14:7(8) | 6:0(7) | 6912 | 22528 | 0.8 |

It is clear from table 5.1 that this architecture produces a significant reduction in LUT size and memory requirements when compared to published versions of PWL and PWP solutions proposed by other researchers (see section 4.5).

Further improvements to this architecture are possible and the structure can be optimised for implementation on an FPGA. Although the results in table 5.1 indicate that the $7:16::7:9$ configuration is the most efficient in terms of total memory used, it does not make the most efficient use of resources when implemented on an FPGA. In Chapter 4 the LUTs have been implemented using dedicated BRAM (Block RAM) elements that are embedded in the FPGA fabric. These BRAM cells have a granularity of 18k bits for Xilinx Spartan3 devices and 9k bits for more modern Spartan6 devices. The optimal implementation above uses just 43% of a BRAM's available capacity to implement LUT1 and less than 19% for LUT2. Although significant numbers of BRAM cells exist on modern FPGA devices they still represent a limited resource. The solution proposed here is to exploit, where possible, the Distributed RAM elements available within the Configurable Logic Block (CLB) of the FPGA fabric. The use of distributed RAM in the programmable fabric of the FPGA devices is more efficient in modern devices due to the change in the underlying slice architecture and is achieved using Xilinx CORE generator. In Spartan3 and Virtex2 devices the distributed RAM elements have dimensions of $16 \times 1$ bits. In more modern devices (such as the Spartan 6 and Virtex 6) the size of distributed RAM elements has increased to $64 \times 1$ bits. Hence it is now possible to embed four times as much memory in a programmable slice. Modern devices also have a substantially increased slice capacity. This is clearly a more efficient use of the available programmable resources and benefits our proposed approach substantially. These memory blocks have a much finer granularity and are embedded in the configurable logic fabric. From table 5.1 it is observed that using an $8:15$ partition for LUT1 would increase the utilisation of a BRAM to 85% while reducing the size of LUT2 to just 1728 bits. This would result in just 9.4% utilization of a second BRAM for LUT2 on a Spartan 3 device. The inefficient utilisation of BRAM for LUT2 indicates that alternative implementations could be more effective.

### 5.2.1 Symmetry

It has been observed that when an increasing number of PWL segments are used to approximate the normalised Lin2Log function, the symmetrical characteristics of the residual error curves generated for each segment increases. A simple differentiation of the curve shows how this effect increases as the number of segments increase. To prove this mathematically differentiation of each error curve is calculated. Figure 5.1 shows a right angled triangle with 3 points $A(x2, y2)$, $B(x2, y1)$ and $C(x1, y1)$.

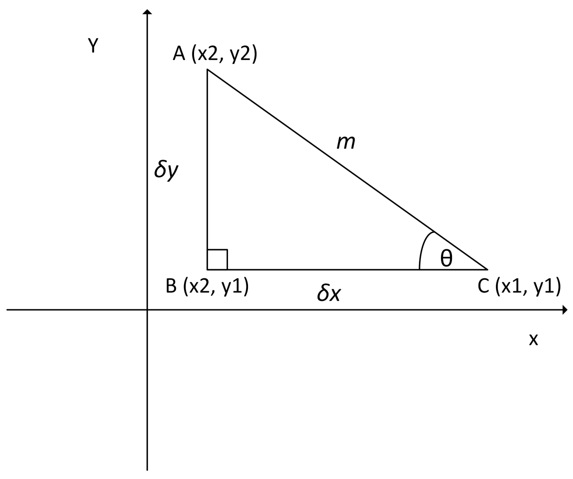


Figure 5.1: Right-angled triangle on cartesian coordinates.

Slope $(m)$ in this triangle is defined as

$$m = \frac{\delta y}{\delta x} \tag{5.1}$$

where

$$\delta y = y2 - y1 \tag{5.2}$$

and

$$\delta x = x2 - x1, \tag{5.3}$$

Hence slope from equation (5.1) becomes

$$m = \frac{y2 - y1}{x2 - x1}. \tag{5.4}$$

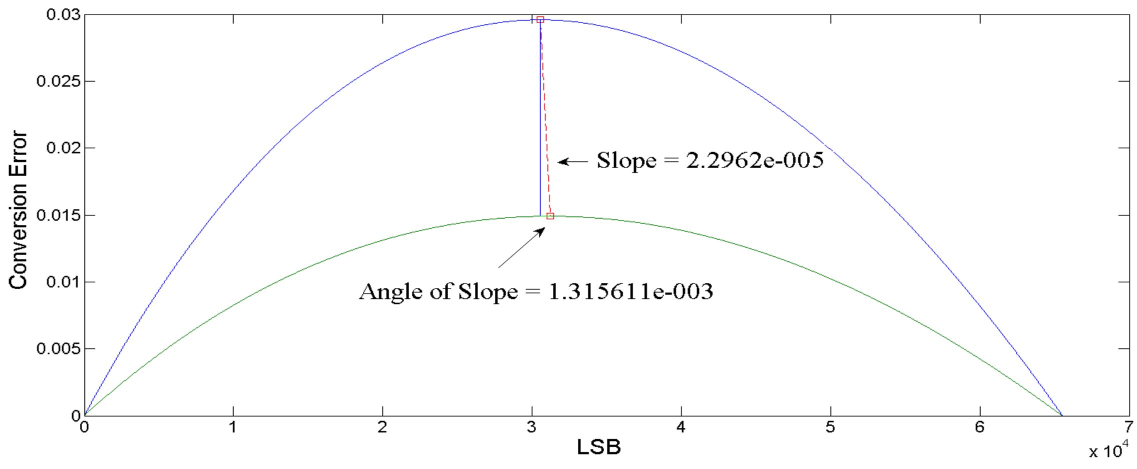
Angle of slope $(\theta)$ in figure 5.1 is defined as

$$tan(\theta) = m = \frac{y2 - y1}{x2 - x1}, \tag{5.5}$$

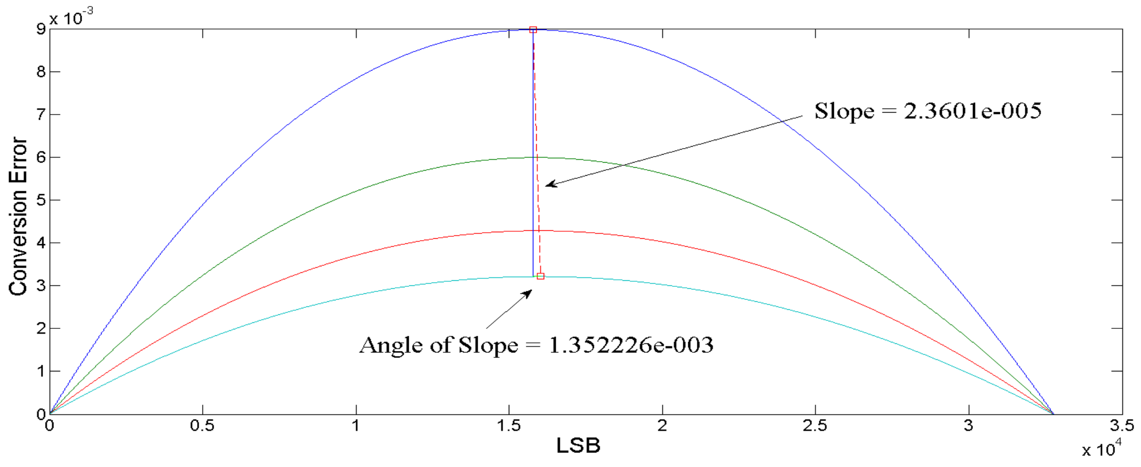
where $\theta$ can be written as

$$\theta = tan^{-1}(\frac{y2 - y1}{x2 - x1}). \tag{5.6}$$

However in figure 5.1 $x1$ is greater than $x2$. As $x1$ approaches $x2$, the magnitude of slope and angle of slope ($\theta$) will increase. This is shown in simulations done in MATLAB, that is as symmetry increases in error curves, the magnitude of slope and angle of slope ($\theta$) increases and vice versa. Figures 5.2 to 5.3 show an example of increase in symmetry where straight line is used as reference for ideal line and dotted line is approximated between highest points on error curves. Two to sixteen error curves are shown in these figures, which proves that with more number of error curves a more symmetrical normalised error curve is produced.

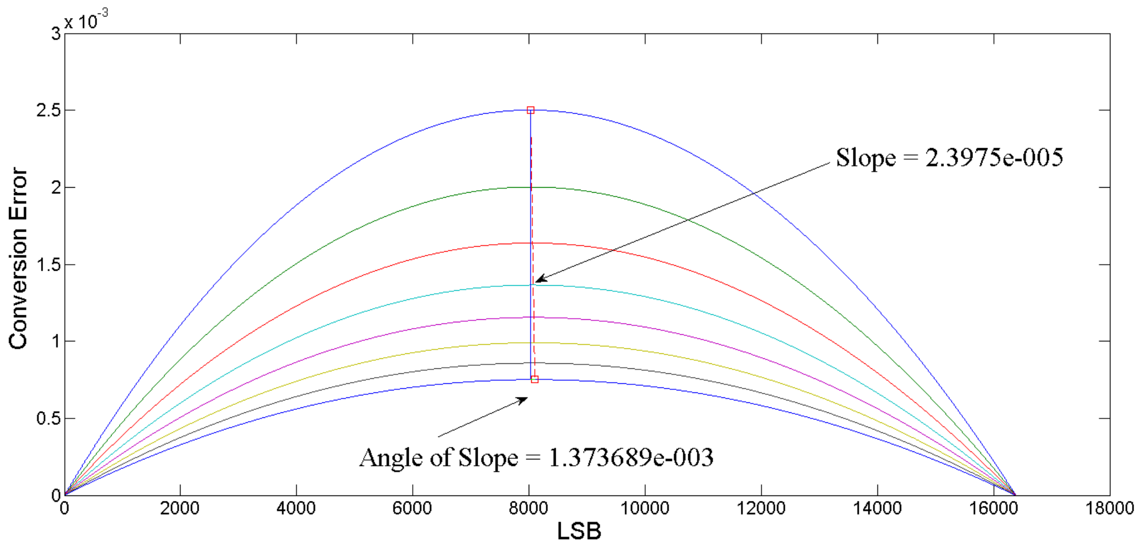


(a)

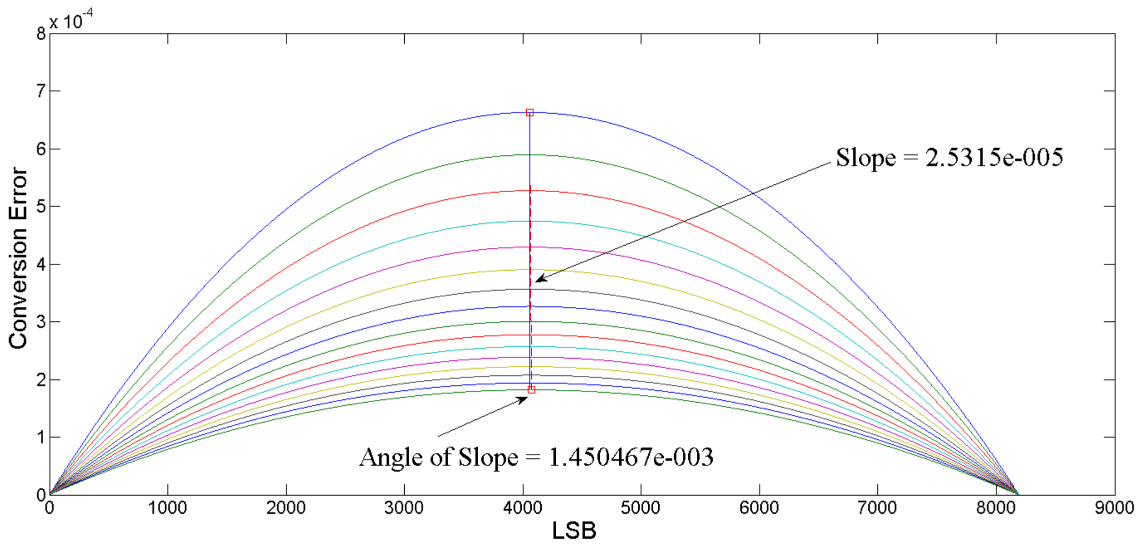


(b)

Figure 5.2: 1st-stage errors showing magnitude of slope and angle of slope, when using (a) 1 MSB and 16 LSB (b) 2 MSB and 15 LSB.



(a)



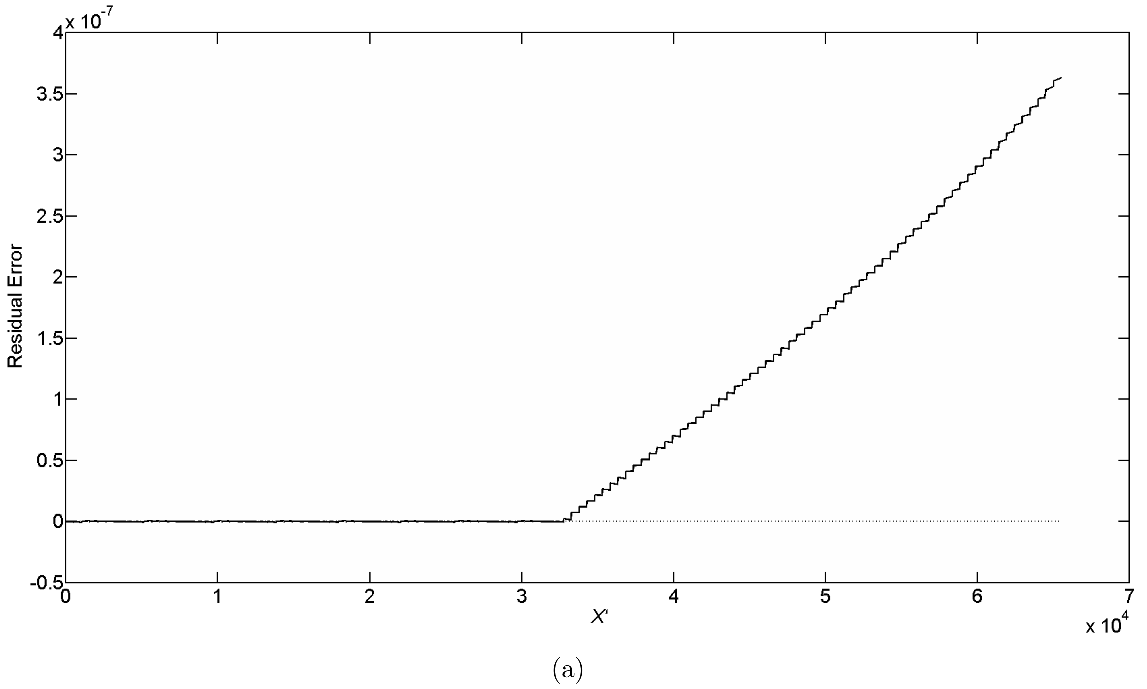
(b)

Figure 5.3: 1st-stage errors showing magnitude of slope and angle of slope, when using (a) 3 MSB and 14 LSB (b) 4 MSB and 13 LSB.

### 5.2.2 LUT2 size reduction

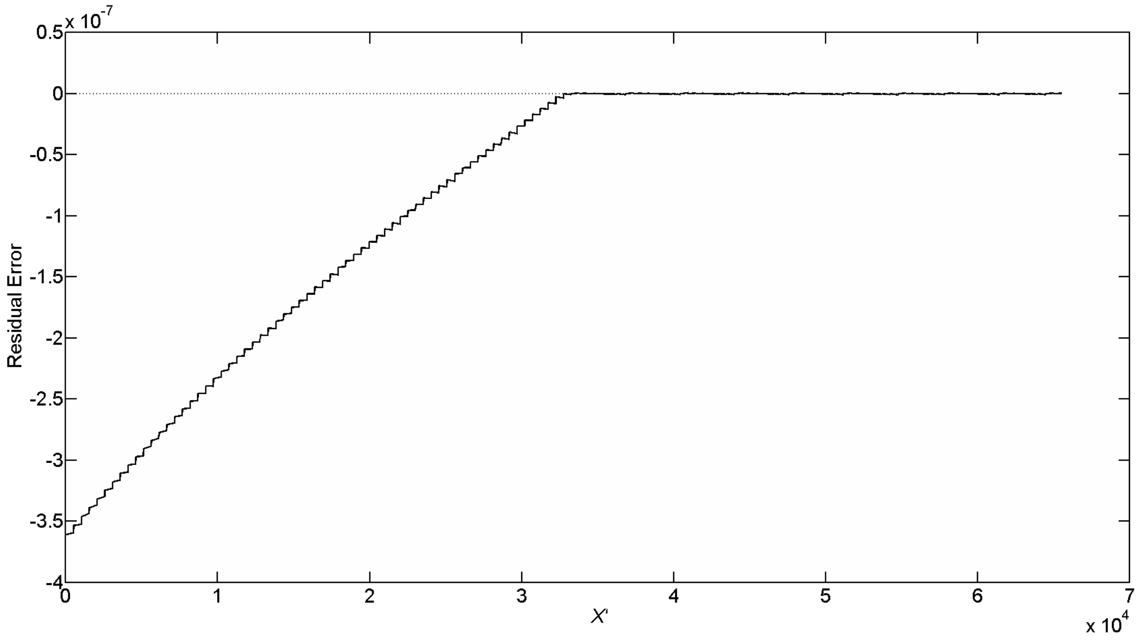
A quick glance at the normalised error curve stored in LUT2 and shown in figure 4.3a indicates that it has a significant component of symmetry about its apex. and is significantly more symmetrical than the Mitchell error curve. This symmetrical property could be used to reduce the size of LUT2 by a factor of 2. Such techniques have frequently been used in the past to reduce the size of LUTs for sine and cosine functions in Direct Digital Synthesis (DDS) by exploiting their (obvious) symmetrical properties.

However, analysis of the curve in figure 4.3a shows that it is not completely symmetrical. In the begining an experiment is performed in which a left and right halves of the normalised error curve (figure 4.3a) are superimposed on other halves. The error produced by this approximation is shown in figure 5.4. This approximation is improved further by using the average of the left and right halves of the curves to generate a symmetrical composite error curve, resulting in the smallest residual error (figure 5.5) between the normalised error curve and the symmetrical approximation to it. However, when this simple symmetrical approximation of the normalised error curve is used the accuracy of the conversion algorithm degrades to less than 23 bits of fractional precision, shown in table 5.2. The table 5.2 MATLAB simulation plots are shown in appendix C.



(a)

Figure 5.4: Residual error produced after superimposing (a) left half of the normalised curve on the other half



(b)

Figure 5.4: Residual error produced after superimposing (b) right half of the normalised curve on the other half

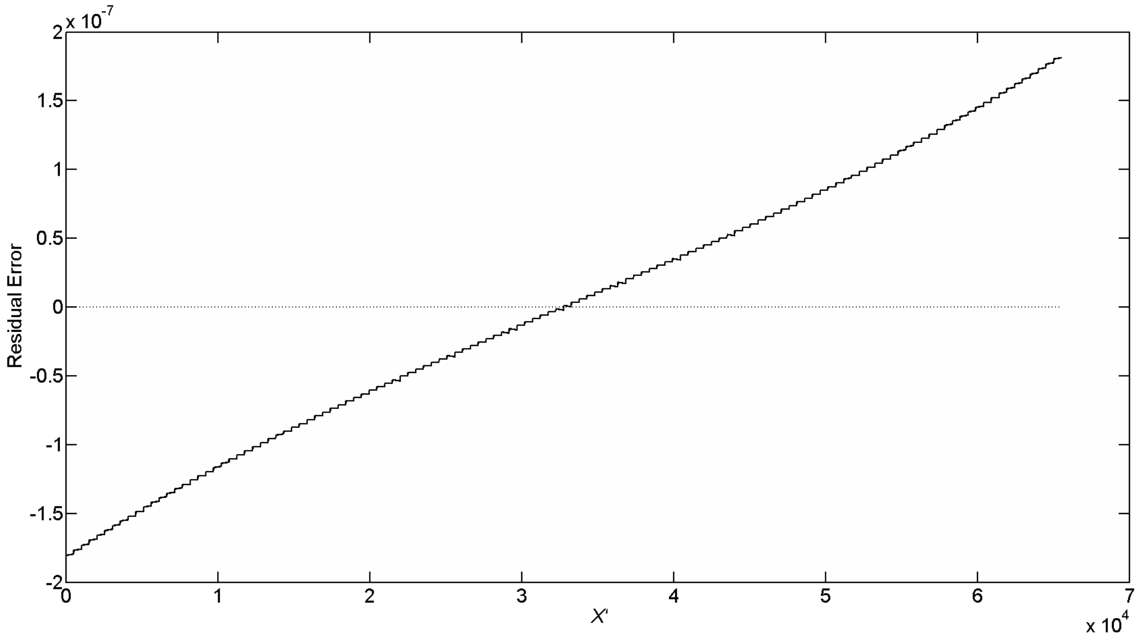


Figure 5.5: Residual error produced after approximating the normalised curve using a symmetrical approximation.

Table 5.2: LUT coefficient bits

| Configuration | LUT1(a=25, b=18, s=18) | | | LUT2(a'=18, b'=9) | | | Total Bits | RMS ($\times 10^{-7}$) |
|---|---|---|---|---|---|---|---|---|
| | Add Bits | Data Bits | LUT1 Bits | Add Bits | Data Bits | LUT2 Bits | | |
| 7:16::7:9 | 22:16(7) | 15:0(16) | 7808 | 15:10(6) | 8:0(9) | 1728 | 9536 | 2.62 |
| 8:15::6:9 | 22:15(8) | 14:0(15) | 15616 | 14:10(5) | 8:0(9) | 864 | 16480 | 1.64 |
| 8:15::7:8 | 22:15(8) | 14:0(15) | 15616 | 14:9(6) | 7:0(8) | 1728 | 17344 | 1.24 |
| 8:15::8:7 | 22:15(8) | 14:0(15) | 15616 | 14:8(7) | 6:0(7) | 3456 | 19081 | 1.03 |

Further analysis of the residual error curve (figure 5.5) shows that it also has symmetrical properties and hence an approximation to the residual error can also be stored in the same LUT as the symmetrical approximation to the normalised error curve. When combined with the symmetrical error using the circuit shown in figure 5.6, the remaining error in the normalised error curve (shown in figure 5.7) is significantly reduced and if this approximation is used then the maximum error produced by whole conversion algorithm is less than 1ULP (Unit of Last Place) for a 23-bit fractional input, that is $1.19 \times 10^{-7}$.

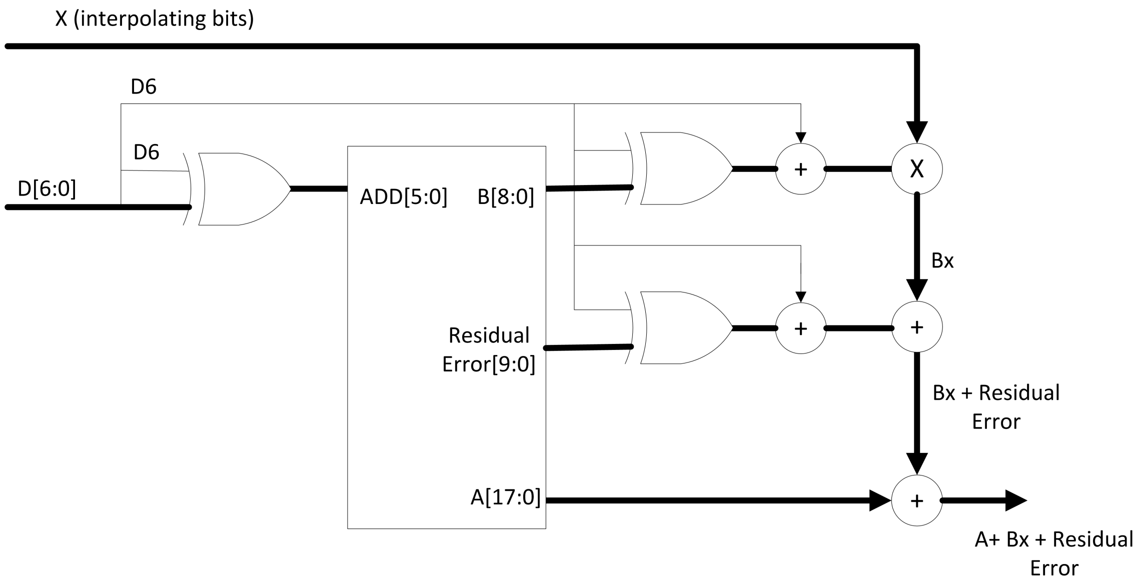


Figure 5.6: Improved LUT2 architecture.

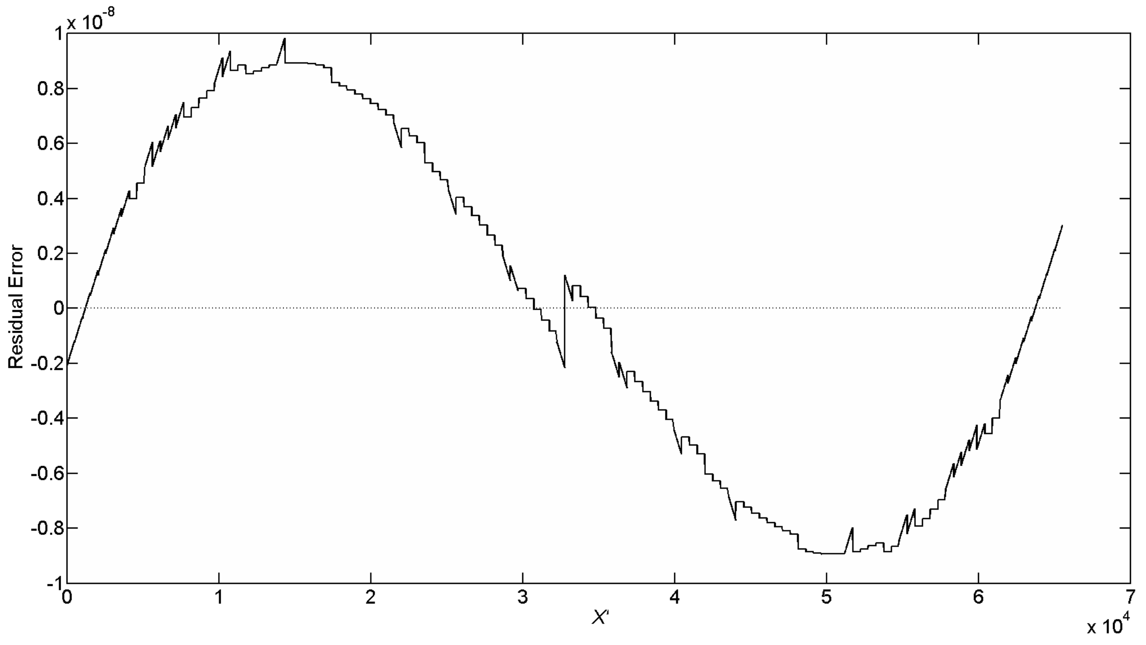
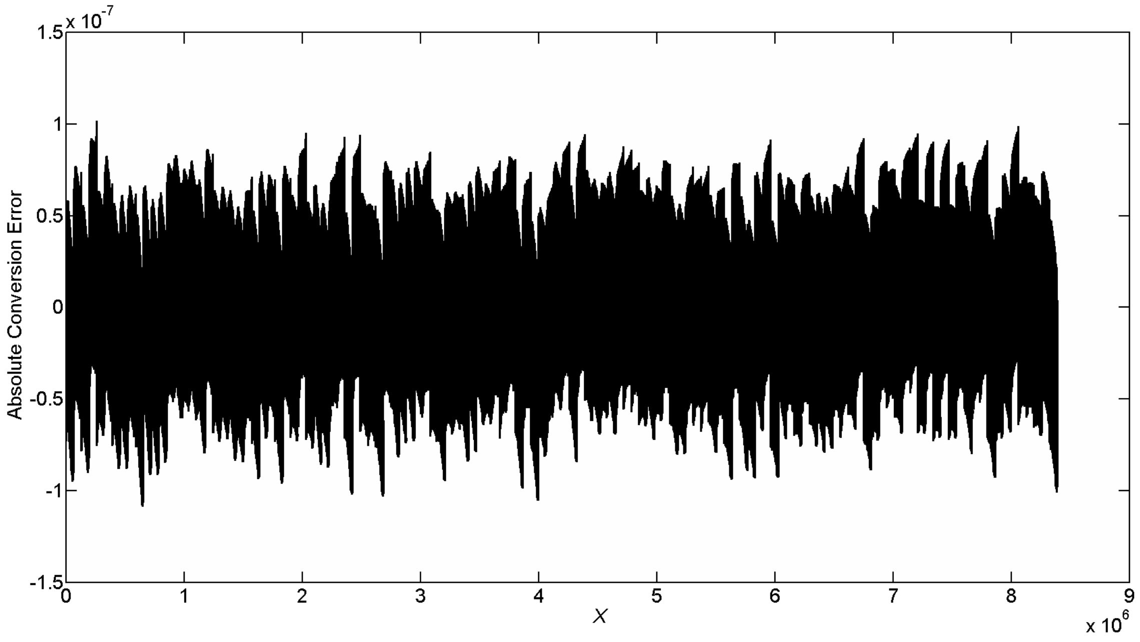


Figure 5.7: Residual error in normalised error curve approximation.

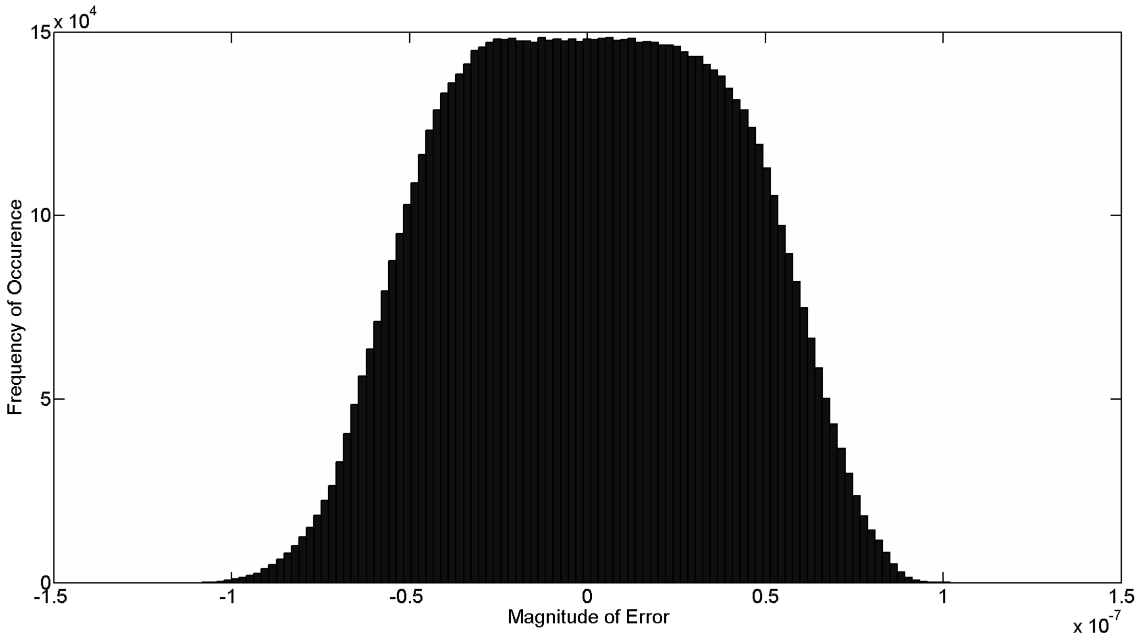
Accuracy achieved when normalised error curve symmetrical property is combined with the symmetrical residual error (5.7) is shown in table 5.3 and their respective MATLAB simulation plots are shown in appendix D. Figure 5.8a shows overall error when first and reduced second stages are combined and figure 5.8b shows even distribution of magnitude of overall error on a histogram.

Table 5.3: LUT coefficient bits

| Config uration | LUT1(a=25, b=18, s=18) | | | LUT2(a'=18, b'=9) | | | Total Bits | RMS ($\times 10^{-7}$) |
|---|---|---|---|---|---|---|---|---|
| | Add Bits | Data Bits | LUT1 Bits | Add Bits | Data Bits | LUT2 Bits | | |
| 7:16::7:9 | 22:16(7) | 15:0(16) | 7808 | 15:10(6) | 8:0(9) | 2368 | 10176 | 1.05 |
| 8:15::6:9 | 22:15(8) | 14:0(15) | 15616 | 14:10(5) | 8:0(9) | 1184 | 16800 | 0.9 |
| 8:15::7:8 | 22:15(8) | 14:0(15) | 15616 | 14:9(6) | 7:0(8) | 2368 | 17984 | 0.9 |
| 8:15::8:7 | 22:15(8) | 14:0(15) | 15616 | 14:8(7) | 6:0(7) | 4736 | 20354 | 0.9 |



(a)



(b)

Figure 5.8: (a) Overall error obtained in approximation. (b) Histogram of approximated error distribution.

## 5.3 Implementation

The algorithm [11] with reduced memory has been implemented using the Xilinx ISE Design suite 13.2 on Xilinx XC3S200 Spartan3 and XC6SLX16 Spartan6 FPGA devices. The architecture design for the proposed algorithm [11] gives the choice between minimum hardware usage and maximum operating frequency. In the minimum pipelining architecture of the proposed algorithm [11], the multipliers are made with zero latency stage limiting the complete circuit with lower operating frequency but with minimum hardware usage to perform the conversion. To increase the operating frequency of complete circuit, a maximum pipelining architecture is shown where the multiplier with three stages (i.e. the optimum stages for pipelining) are used. The post-implementation results of both the architectures are shown in table 5.4. The FPGA design top level diagram for maximum and minimum pipelining of the proposed algorithm [11] architectures are shown in figures 5.9 and 5.10.

Table 5.4: Device Utilisation Statistics (LUT2 with additional adder and xor gates)

| Spartan3 utilisation | BRAM | HMUL | CLB slices | speed,MHz | Latency,clks |
|---|---|---|---|---|---|
| min pipelining | 1 | 3 | 187 | 33.42 | 2 |
| max pipelining | 1 | 3 | 317 | 93.34 | 8 |
| Spartan6 utilisation | BRAM | DSP48 | CLB slices | speed,MHz | Latency,clks |
| min pipelining | 1 | 3 | 49 | 46.14 | 2 |
| max pipelining | 1 | 3 | 73 | 110.5 | 8 |

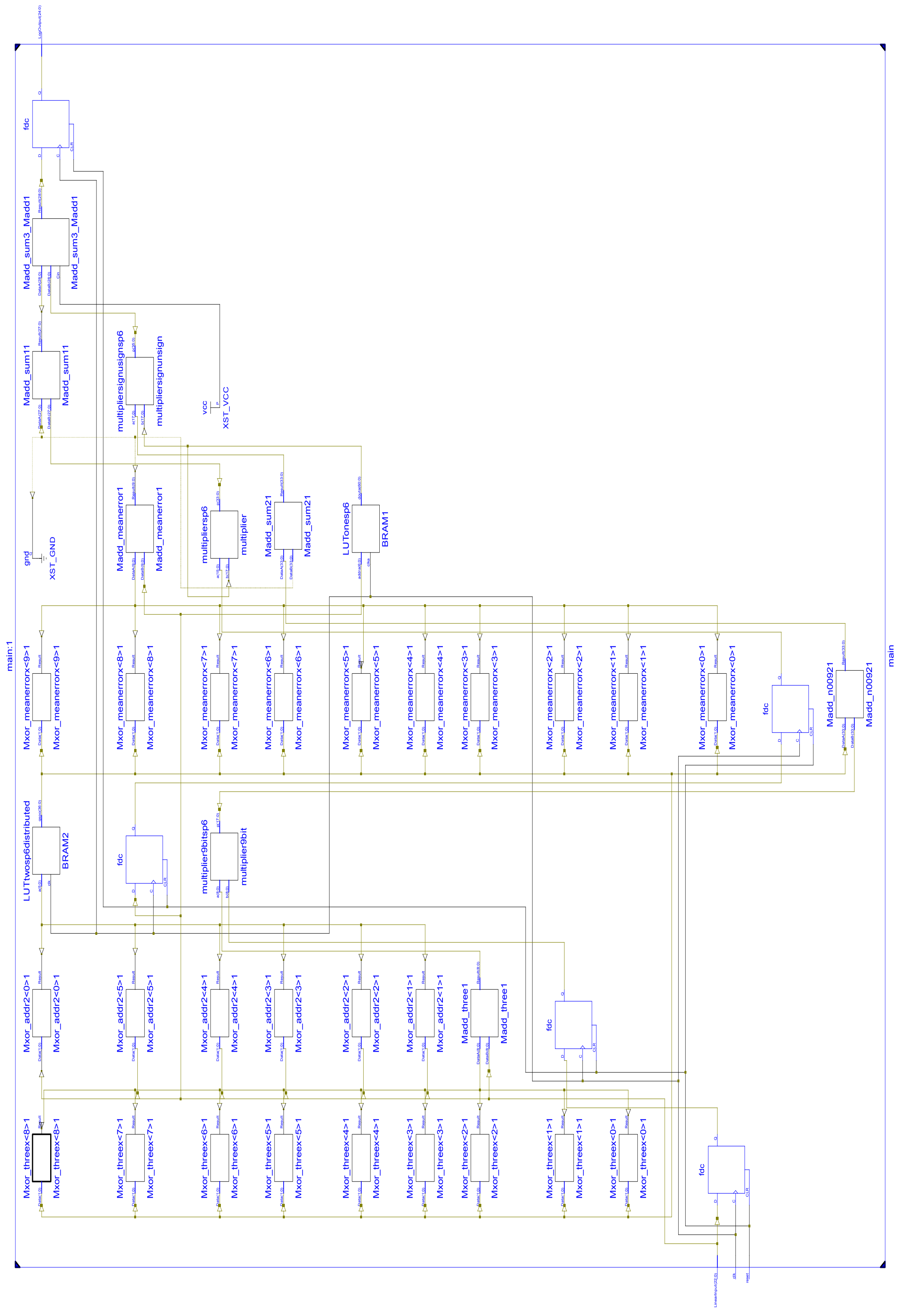


Figure 5.9: Spartan 6 based minimum pipelining design of the novel algorithm

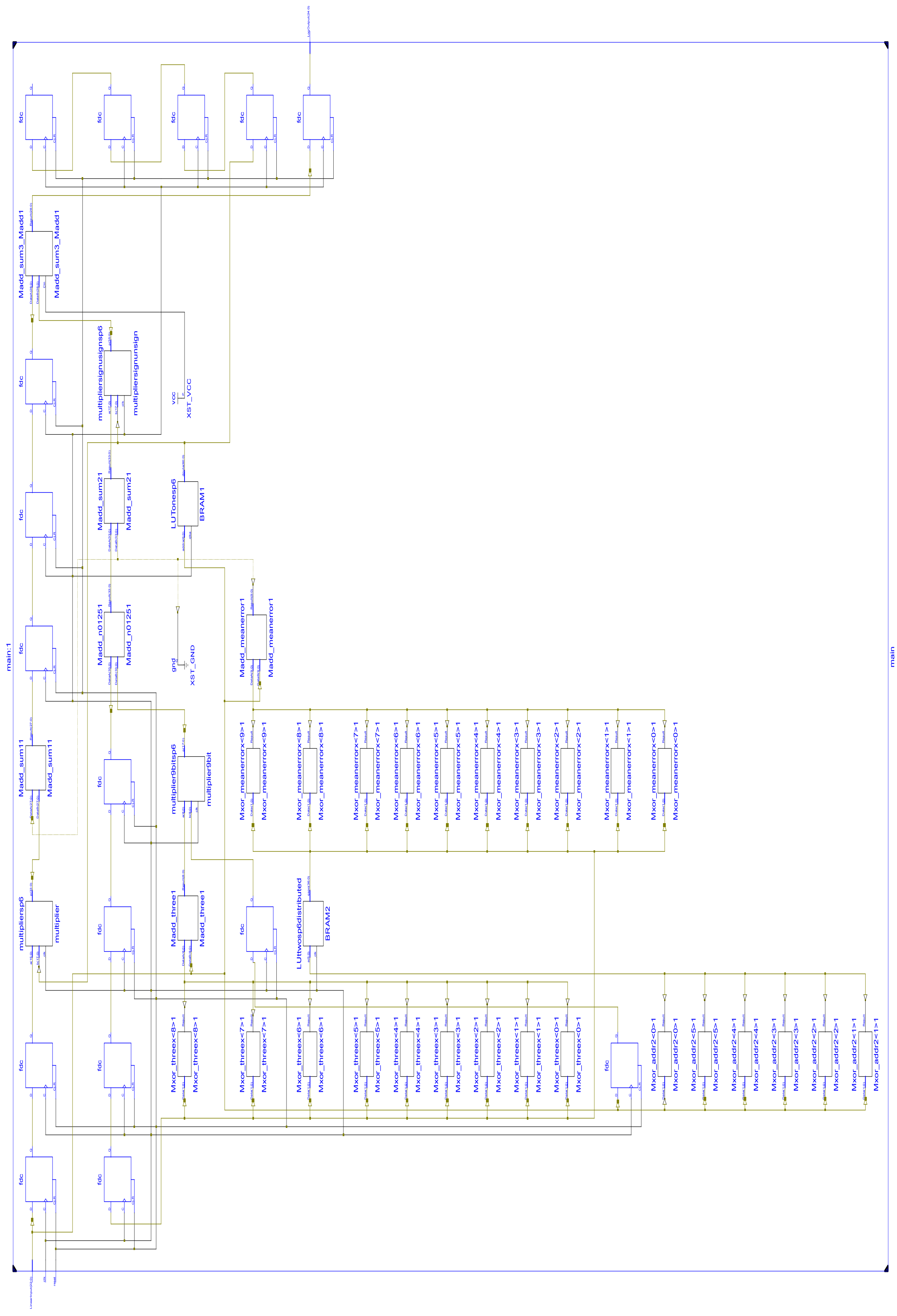


Figure 5.10: Spartan 6 based maximum pipelining design of the novel algorithm

From table 5.4 is observed that when LUT2 is implemented using reconfigurable fabric it needs just 187 slices on a Spartan 3 device and 49 slices on Spartan 6 device, when no pipelining is used. In case of maximum pipelining 317 slices on a Spartan 3 device and 73 slices on a Spartan 6 device are used. This modest increase in the number of slices needed in the design is offset by the fact that the architecture now uses only one BRAM instead of two. The size of the LUT is reduced to just 1184 bits or reduced by 32% of the original LUT2 at the cost of additional adder and XOR gates at the LUT inputs and outputs (see figure 5.6).

Device utilisation statistics when LUT2 is implemented only on distributed RAM (using a Xilinx CORE generator see section 5.2 ) without using any additional circuitry (adder and xor gates) is shown in table 5.5. The additional logic (adder and xor gates) shown in table 5.4 reduces the number of CLB slices needed to implement LUT by a further 4% when compared to just using the CLBs or slices for the LUT.

Table 5.5: Device Utilisation Statistics (LUT2 without additional adder and xor gates)

| Spartan3 utilisation | BRAM | HMUL | CLB slices | speed,MHz | Latency,clks |
|---|---|---|---|---|---|
| max pipelining | 1 | 3 | 331 | 127.7 | 8 |
| Spartan6 utilisation | BRAM | DSP48 | CLB slices | speed,MHz | Latency,clks |
| max pipelining | 1 | 3 | 76 | 1589.9 | 8 |

Previously published [118] and [119] used Xilinx Virtex-II family to compare logic utilisation. Although a different device, the underlying architecture of the slices in both devices is the same. Hence as expected, the table 5.6 shows that there is almost no difference in the number of logic slices required by both Spartan 3 (see table 5.5) and Virtex II devices. The only significant change is the substantial increase in the operating frequency of the algorithm on a Virtex II device due to the fabrication technology used (not the architecture).

Table 5.6: Device Utilisation Statistics (Reduced LUT2 architecture implemented)

| Device | BRAM | HMUL | slices | speed,MHz | Latency,clks |
|---|---|---|---|---|---|
| V2-2000 | 1 | 3 | 351 | 264.03 | 8 |
| V2-6000 | 1 | 3 | 351 | 264.03 | 8 |

## 5.4 Results

In table 5.7 results from table 5.4 are compared with recent work [119] [118] for conversion with 23 bits of fractional precision (or equivalent to an accuracy of 1.19 $\times 10^{-7}$) indicating that similar accuracies are achieved with less BRAM. In table 5.7 F stands for faithful rounding units and B stands for BTFP (Better Than Floating Point) units as described in [118]. This comparison shows there is a reduction in the number of BRAM of 50% offset by a modest increase in the number of slices from the algorithm [10] presented in Chapter 4.

BRAM has a fixed capacity (18 kbits) and, even on modern FPGA devices, are a limited resource. For example the test device used in this chapter is a Spartan3 XC3S200 and has only 12 BRAMs. In contrast it has 1,920 logic slices available. Hence the 30% increase in slices (from 232 to 317 on a Spartan3) actually represents an increase of 85 slices or a 4.5% increase of the total slices available on the device. This modest increase is offset by a 50% reduction in the number of BRAMs used in the design (a reduction of 8.3% in the BRAMS required  a much more limited resource). Therefore this represents a credible design choice.

Table 5.7: Utilization Comparison

| Source | Device | BRAM | HMUL/DSP48 | slices |
|---|---|---|---|---|
| [119] | Virtex II XC2V2000 | 24 | 0 | 163 |
| [118]F* | Virtex II XC2V6000 | 2 | 3 | 286 |
| [118]B** | Virtex II XC2V6000 | 2 | 3 | 327 |
| Table 4.4 | Spartan 3 XC3S200 | 2 | 3 | 232 |
| Table 5.4 | Spartan 3 XC3S200 | 1 | 3 | 317 |
| Table 4.4 | Spartan 6 XC6SLX16 | 2 | 3 | 55 |
| Table 5.4 | Spartan 6 XC6SLX16 | 1 | 3 | 73 |

F* Faithful rounding units, B** BTFP (Better Than Floating Point)

## 5.5 Conclusion

This chapter describes a novel binary Linear to Log (Lin2Log) conversion algorithm [11] that has been optimised for implementation on an FPGA. The novel algorithm [11] shows improvement to the algorithm [10] proposed in Chapter 4 by a 50% reduction in the total number of BRAM used. The algorithm [11] is based on a two-stage approximation and uses fewer FPGA resources i.e. number of slices and multipliers when compared with other piecewise linear (PWL) and non-uniform piecewise polynomial (PWP) architectures that have been proposed recently.

This chapter exploits the symmetrical properties of a normalised error curve to reduce the size of the second-stage LUT by 32%. The MATLAB simulations and analysis of different configurations of data bits and resolution of coefficient bits stored in memory are provided in this chapter. The proposed algorithm [11] uses additional XOR gates allowing it to be implemented more efficiently using the combinatorial logic available (achieved by using distributed RAM) in the reconfigurable fabric instead of a second BRAM. The architecture presented in this chapter achieves 23 bits of fractional precision while operating at a maximum frequency of 93 and 110 MHz when implemented on Xilinx Spartan3 and Spartan6 devices respectively and requires just one 18k bit Block RAM (BRAM) element for the first-stage Look-Up Table (LUT).

# Chapter 6

# FPGA based Correlation Velocimetry System

## 6.1 Introduction

This chapter provides a description of an application of FPGA and logarithmic processing using correlation techniques. Correlation is a technique which is used to show the interdependence or similarity between two or more signals. This technique is widely used in digital signal processing applications such as image processing for robotic vision, remote sensing by satellite and climatology applications [123].

Applications such as radar and sonar systems use correlation techniques for finding a specific object in space. A problem faced by these systems was accuracy in determining the location of an object in space. Reflected signals coming from an object in space are highly corrupted with noise, creating the problem of object presence. By using correlation techniques this problem can be solved, where received signal is correlated with the transmitted signal. In digital communication, applications like spectral analysis and statistical estimation are performed using correlation technique [124] to find the original signal buried in noise by comparing it with the original signal.

A velocimetry system using cross-correlation technique is used to calculate the speed of moving particles in a pipeline. In previously published [125–129], the velocimetry used extensive computations and complex architectures. This chapter presents a novel architecture using LNS in the velocimetry system to simplify the computations and architecture required in calculating the speed of moving particles in a pipeline. The velocimetry application is used in this research as it is a real-time based application, requiring limited accuracy with a high dynamic range (see section 1.1).

The chapter begins with an overview of correlation techniques. In section 6.2, the cross-correlation principle using FFT (Fast Fourier Transform) and incremental correlation algorithms are described. Section 6.3 describes an overview of published velocimetry systems, describing the application to velocimetry systems with their architecture and performance achieved. Section 6.4 presents an FPGA-based prototype velocimetry system. The algorithm and a hardware implementation of the prototype device is presented in sections 6.5 and 6.6. Section 6.7 shows the results obtained after implementing the system on an FPGA and compares performance with existing solutions developed at Kent (also known as Kent Method).

## 6.2 Correlation

In signal processing correlation technique is divided into two types - cross-correlation and autocorrelation techniques using discrete-time input/ouput signals. In a cross-correlation two independent discrete-time signals are compared and in autocorrelation a discrete-time signal is compared with a delayed version of itself. Autocorrelation is a special case of cross-correlation in which the discrete-time signal is correlated with itself to find information missing from the signal such as frequency and periodic nature of signal.

### 6.2.1 Cross-correlation

Cross-correlation is performed to find similarity or interdependence between two or more discrete-time independent signals. Cross-correlation is used for many applications such as tomography [130–134], pattern distribution and detection [135–138], velocimetry [139–143], digital image processing [144–149] and stereovision [150–154]. Mathematically the cross-correlation ($r$) between two discrete-time independent signals is represented as

$$r_{x_1x_2} = \frac{1}{N}\sum_{n=0}^{N-1} x_1(n)x_2(n) \tag{6.1}$$

where $x_1(n)$ and $x_2(n)$ are two independent periodic signals, each with $N$ period. Similarly autocorrelation of $x_1(n)$ and $x_2(n)$ signals can be found out as shown in 6.2 and 6.3.

$$r_{x_1x_1} = \frac{1}{N}\sum_{n=0}^{N-1} x_1^2(n) \tag{6.2}$$

$$r_{x_2x_2} = \frac{1}{N}\sum_{n=0}^{N-1} x_2^2(n) \tag{6.3}$$

To keep the cross-correlation in the normalised range (i.e. from $-1$ to $+1$), the cross-correlation between the two signals is divided by autocorrelation of each signal. Mathematically normalisation of cross-correlation is represented as

$$\rho_{x_1x_2} = \frac{r_{x_1x_2}}{\sqrt{r_{x_1x_1}r_{x_2x_2}}}. \tag{6.4}$$

$$\rho_{x_1x_2} = \frac{r_{x_1x_2}}{\frac{1}{N}\sqrt{\sum_{n=0}^{N-1} x_1^2(n)\sum_{n=0}^{N-1} x_2^2(n)}}. \tag{6.5}$$

where $\rho_{x_1x_2}$ is the cross-correlation coefficient. The +1 and -1 means complete correlation of these two signals in phase and antiphase respectively. The value 0 in the correlation coefficient shows no similarity between two signals or they are completely independent of each other.

### 6.2.2 Fast cross-correlation

In section 6.2.1 cross-correlation is defined in the time domain. When cross-correlation is performed in real-time application, due to the constant stream of data in cross-correlation, computation of its sequence becomes large. To make computations simple cross-correlation is computed using the FFT (Fast Fourier Transform). Signals used in computing cross-correlation are multiplied in the frequency domain using the DFT (Discrete Fourier Transform) and then their product is converted back into the time domain using the inverse of DFT. Fast cross-correlation mathematically can be represented as

$$r_{x_1x_2} = \frac{1}{N} F_D^{-1}[X_1^*(k)X_2(k)] \tag{6.6}$$

where $X_1(k)$ and $X_2(k)$ are the DFTs of $x_1(n)$ and $x_2(n)$ periodic signals, each with N period. $F_D^{-1}$ represents the inverse of DFT. For a cross-correlation using FFT, multiplication of one signal with another signal conjugate is required, which is denoted as $'*'$ in equation 6.6 [155].

### 6.2.3 Incremental cross-correlation

Another way of speeding up the computing of cross-correlation is by adding new cross-correlation values to previous values and then subtracting the old cross-correlation values from it. By using this approach computations are reduced to a single computation instead of computing the complete cross-correlation sequence every time as described in section 6.2.1. Assuming $N$ number cross-correlation, this approach gives correct values after $N - 1$ values of cross-correlation [123].

$$\begin{aligned} new \;\; value = previous \;\; value + \tfrac{1}{N}(product \;\; of \;\; the \;\; two \;\; new \;\; data) \\ -\tfrac{1}{N}(product \;\; of \;\; the \;\; first \;\; two \;\; data) \end{aligned} \tag{6.7}$$

## 6.3 Velocimetry System

One of the important applications of correlation in industrial applications, is to measure the velocity of flow of particles, strips and conveyor belts. A velocimetry system provides industries with a more reliable and accurate system for measuring the speed of moving strips, pneumatic particulates, rotating machinery and many more. In a coal-fired power station, speed measuring of its pipeline is essential in order to have a uniform flow of coal particulates, which are responsible for complete power station efficiency and can contribute to emission reduction [125].

A velocimetry system may use different methods to calculate the speed, one of which is cross-correlation. In a velocimetry system, different types and combinations of sensors are used in the cross-correlation method, like electrostatic, ultrasonic and radiometric [156]. In 2010 [127] shows the use of electrostatic sensors in measuring

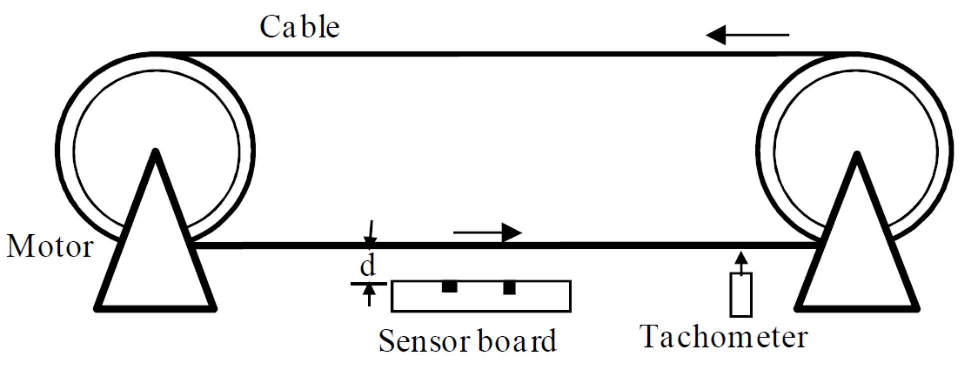


Figure 6.1: Cable speed test rig.

the speed of pneumatically conveyed particles. The authors compare circular and rod electrodes for measuring the speed of coal and biomass particles. The authors claim that rod sensors give better signals or higher cross-correlation coefficients in a comparison with circular electrodes. However, circular electrodes give signals produced around the walls of a pipeline, whereas the rod electrodes generate signals around the local area where they are installed instead of measuring particles everywhere around the pipeline.

A method to measure strip speed using electrostatic signals was proposed in [128]. In this, speed of surface is calculated by knowing the distance between electrodes and time taken by a signal to pass these electrodes. The tests were conducted on a prototype system where a cable is rotated with the help of a motor, shown in figure 6.1. Absolute accuracy and repeatability tests were performed on the speed measuring system. The test on the system shows a relative error of $\pm1.5\%$ for the speed range of 0 to 10 $ms^{-1}$. For the speed of a system between 3 to 8 $ms^{-1}$, relative error is further reduced to $\pm0.5\%$.

There are industries where cutting machines are used to cut strips of metal, cables, leather and cotton. This cutting process is of fixed length, which when done with negligence or inaccuracy can result in wastage of a high amount of raw materials. In 2011 [129] presented a non-contact measurement of strip speed using electrostatic sensors in combination with correlation signal processing techniques. The author uses electrodes to measure the moving speed of cable strip. The friction caused by air and strip surface passes a charge to electrodes. Electrodes are set at fixed positions. The first electrode is called upstream and second one is downstream. The downstream electrode transits the time delayed signal, the same as upstream with added noise. The author uses a tachometer as a reference for comparison with the strip speed system designed by him. The comparison shows a relative error and repeatability error of $\pm 1.8\%$ and 2.5% respectively. The speed range of this strip speed measuring system is claimed to be 0.8 to 10 $ms^{-1}$.

With the use of sets of electrostatic sensor arrays and data fusion techniques, measurement of coal and biomass particles in pipelining has been further improved [125, 126]. In [126], the author uses circular and arc shaped electrostatic sensors to measure average and localised speed of coal and biomass particles flowing through the pipeline. An algorithm of an multiple channel cross-correlation velocimetry system was presented in [126]. Multiple values by permutation of cross-correlation coefficient are computed by the velocimetry system. In data fusion techniques, weighted and simple moving averages are used to produce the results with greater accuracy than other proposed methods. This paper uses a dsPIC kit based circuit, which has a sampling frequency of 50 kHz and operates on 2048 samples at each cross correlation coefficient. However, the author does not mention in detail the actual hardware circuit implementation parameters such as area, power, etc.

Another important field for a velocimetry system is to calculate a rotational speed used by rotating machineries such as electric rotating machineries, turbines, propellers and engines. Mathematical modelling for rotational speed measurement while using electrostatic sensors is presented in [157]. The authors in [158] use cross-correlation to find rotational speed in real-time and claim that system performance is dominantly affected by distance between the electrode and rotating object surface. In the speed range of 0 to 3000 revolutions per minute, the maximum error generated by the system is $\pm 2\%$.

## 6.4 FPGA based Velocimetry System

The methods described in section 6.3 used microntrollers [129] or dsPIC kits [126] to find velocity and other parameters of particles in the pipeline. FPGAs in the past have been used for correlation applications. Urena [159] used FPGAs for correlation detection using ultrasonic sensors, showing an improvement on previous transducer electronic systems. The method described in [159], improved the accuracy of TOFs (times of flight) which is a similar approach to radar applications. In this method the correlation between signal sent and echo signal received is found by using a peak detector. In [160], for finding phase only correlation, an image captured by a CMOS camera in FPGAs is processed at a higher/faster speed. The correlation is performed using Fast Fourier Transform and Inverse Fast Fourier Transform. The FPGAs have been also used for 2D and 3D correlation tracking applications [161], real-time stereovision systems [162], real-time correlation on voice signals [163] and embedded systems for comparing finger prints using cross-correlation [164].

In [165], signal processing of a two-channel electrode system has been implemented on an FPGA, where the system monitors the continuous flow rate of pneumatically conveyed particulates by calculating their cross correlation. The authors [165] present a real-time logarithmic based arithmetic circuit which is 1000x times faster than previously published microcontroller based solution. In [165], the circuit is implemented on the Xilinx Spartan3 device operating at the frequency of 50 MHz. The architecture is capable of a delay in range of 0 to 20.48 ms with a resolution of 20.48 $\mu s$ at the sampling rate of 48.8 kHz. The cross-correlation computation in the past has been speeded up by using the FFT (described in section 6.2.2). However, while using the FFT, the conversion of signals from time domain to frequency domain (by using DFT) and their product back into the time domain, makes the hardware circuit highly complex and vast.

For a real-time correlation application, to speed up the computation on the hardware basis, an incremental correlation algorithm is preferred (described in section 6.2.3). Incremental correlation is simpler and easier to implement on hardware. The computation in incremental correlation algorithm is speeded up by implementing the correlation calculation recursively. The calculation is repeated when next sampled data values are arriving; their product is added and the last product of old sampled values are deleted. This calculation after initial set up requires only one multiplication, one subtraction, one addition and one division.

This chapter presents a novel architecture for an FPGA-based real-time correlation application using an incremental correlation algorithm first presented in [165]. The novel architecture presented in this chapter uses the 2 and 4-electrode system first published in [128, 129] and [126] at a higher/faster operating frequency rate.

## 6.5 Velocimetry System Algorithm

In a pipeline, the velocity of particles flowing through it can be measured by using sensors embedded into its walls. With the relative motion in particles, air and pipe wall there is a production of electrostatic charge. This electrostatic charge, measured with electrodes, is preamplified before performing any computations. In [128] and [129], two electrodes with dedicated electric circuits are positioned downstream and upstream (see figure 6.2). The length between two electrodes is fixed and the particle velocity is derived by using

$$V = \frac{L}{T} \tag{6.8}$$

where $V$ is the velocity, $L$ is the centre-to-centre spacing between the two electrodes and $T$ is the time taken by particles to flow from one electrode upstream to the other in the downstream. The time taken by particles can be found by plotting the correlation function of two signals and the highest peak on the plot will denote to the sampled value or time taken by particles to flow from upstream to downstream.

The authors in [128] describe the problems faced in the correlation based velocimetry system such as failing to identify any change in velocity with high accuracy and spurious readings because of irregularity in the cross-section of a pipeline. The author in paper [126] solved this problem by using a set of sensor arrays and data fusion techniques, where the maximum of information is acquired from particulate flow with a high accuracy.

A cross-correlation function using upstream and downstream channels is defined as

$$C_{susd} = \frac{1}{N} \sum_{n=0}^{N-1} su(n-m)sd(n) \tag{6.9}$$

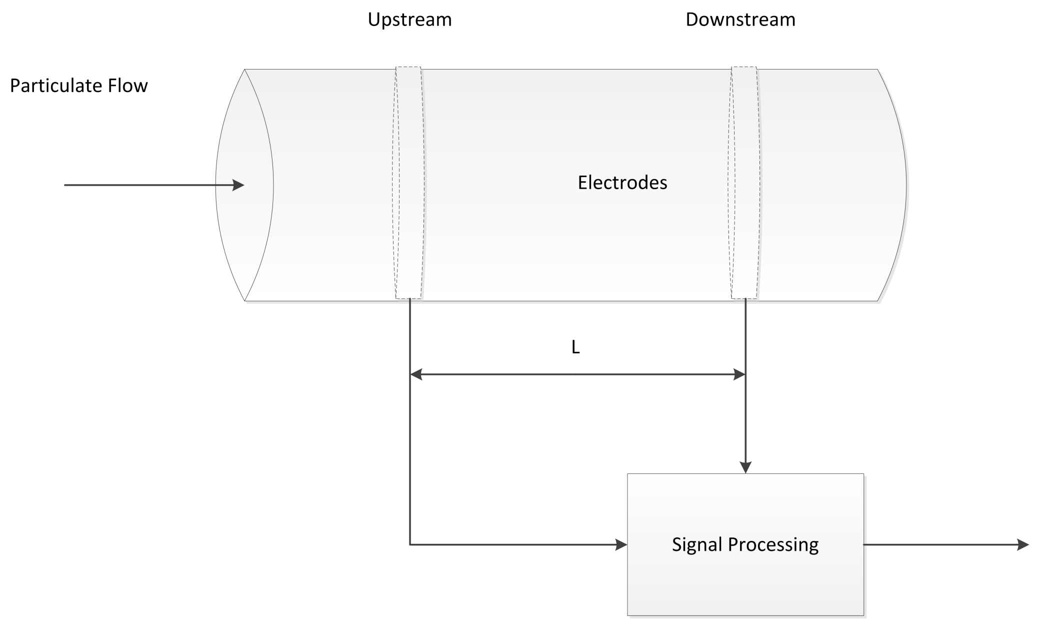


Figure 6.2: 2 channel velocimetry system.

where $C_{susd}$ is the cross-correlation between two signals $su(n)$ (upstream signal) and $sd(n)$ (downstream signal), $m$ is the sampled delay between upstream and downstream and $N$ is the number of samples. The cross-correlation performed in 6.9, when implemented in hardware creates a problem of bottleneck [165]. The bottleneck problem was earlier resolved by doing correlation in a number of sub-sets. The number of sub-sets limits the correlation fluctuation and overall accuracy in the flow rate. The efficient way to address this problem is by using incremental correlation algorithm as every new result is produced at every new sample. The property of incremental calculations of producing new results at every new sample helps the circuit to be a real-time application. Hence equation 6.9 is converted into an incremental correlation algorithm, shown in equation 6.10.

$$\begin{aligned} C_{susd,m,t} = C_{susd,m,t-1} + \tfrac{1}{N}(sd_{0+mT} \times su_0) \\ -\tfrac{1}{N}(sd_{CL+mT} \times su_{CL}) \end{aligned} \tag{6.10}$$

where $T$ is the sampling period, $mT$ is total delayed time for the sample and $CL$ is the correlation length or number of samples. $C_{susd,m,t}$ at time $nt$ is calculated from the previous value $C_{susd,m,t-1}$ (value of correlation at previous sample) by adding the new value of correlation and subtracting the oldest value from it. For normalising the cross-correlation, autocorreation of upstream and downstream signals is computed in an incremental manner as shown in equations 6.11 and 6.12.

$$\sum su_t^2 = \sum su_{t-T}^2 + \tfrac{1}{N}(su_0^2) - \tfrac{1}{N}(su_{512}^2) \tag{6.11}$$

$$\sum sd_{m,t}^2 = \sum sd_{m,t-T}^2 + \tfrac{1}{N}(sd_{0+m,t}^2) - \tfrac{1}{N}(sd_{512+m,t}^2) \tag{6.12}$$

To further simplify normalisation, $C_{susd}$, $su^2$ and $sd^2$ are converted in the log domain from linear domain. The multiplication and division in equation 6.13 are reduced to addition and subtraction, shown in equation 6.14. Hence, it becomes simple to implement these computations on hardware. Such computation also uses fewer hardware resources.

$$\rho_{susd} = \frac{C_{susd}}{\sqrt{su^2 \times sd^2}} \tag{6.13}$$

$$log_2(\rho_{susd}) = log_2(C_{susd}) - \tfrac{1}{2}(log_2(su^2) + log_2(sd^2)) \tag{6.14}$$

## 6.6 Hardware Implementation

The incremental cross-correlation algorithm described in section 6.5 is implemented in hardware by using a 4-channel velocimetry system. A schematic diagram of a 4-channel velocimetry system is shown in figure 6.3, where $X1$, $X2$, $X3$ and $X4$ represent the equally spaced (at a fixed length of $L$) arrays of sensors embedded in a pipeline wall. These sensors in the pipeline wall get electrostatically charged when the particulates in the pipeline flow through them. Cross-correlation computations are performed on six values (by permutation) combinations of the sensors and are represented in figure 6.3 as $r_{x1x2}$, $r_{x1x3}$, $r_{x1x4}$, $r_{x2x3}$, $r_{x2x4}$ and $r_{x3x4}$. The peak values are detected in the cross-correlation computations performed above, denoting the time (sample value) taken by particulates to flow from one sensor to the other. In figure 6.3, the time (sample value) for each of the combination of sensors are represented as $T_{12}$, $T_{13}$, $T_{14}$, $T_{23}$, $T_{24}$ and $T_{34}$. The data fusion block in figure 6.3

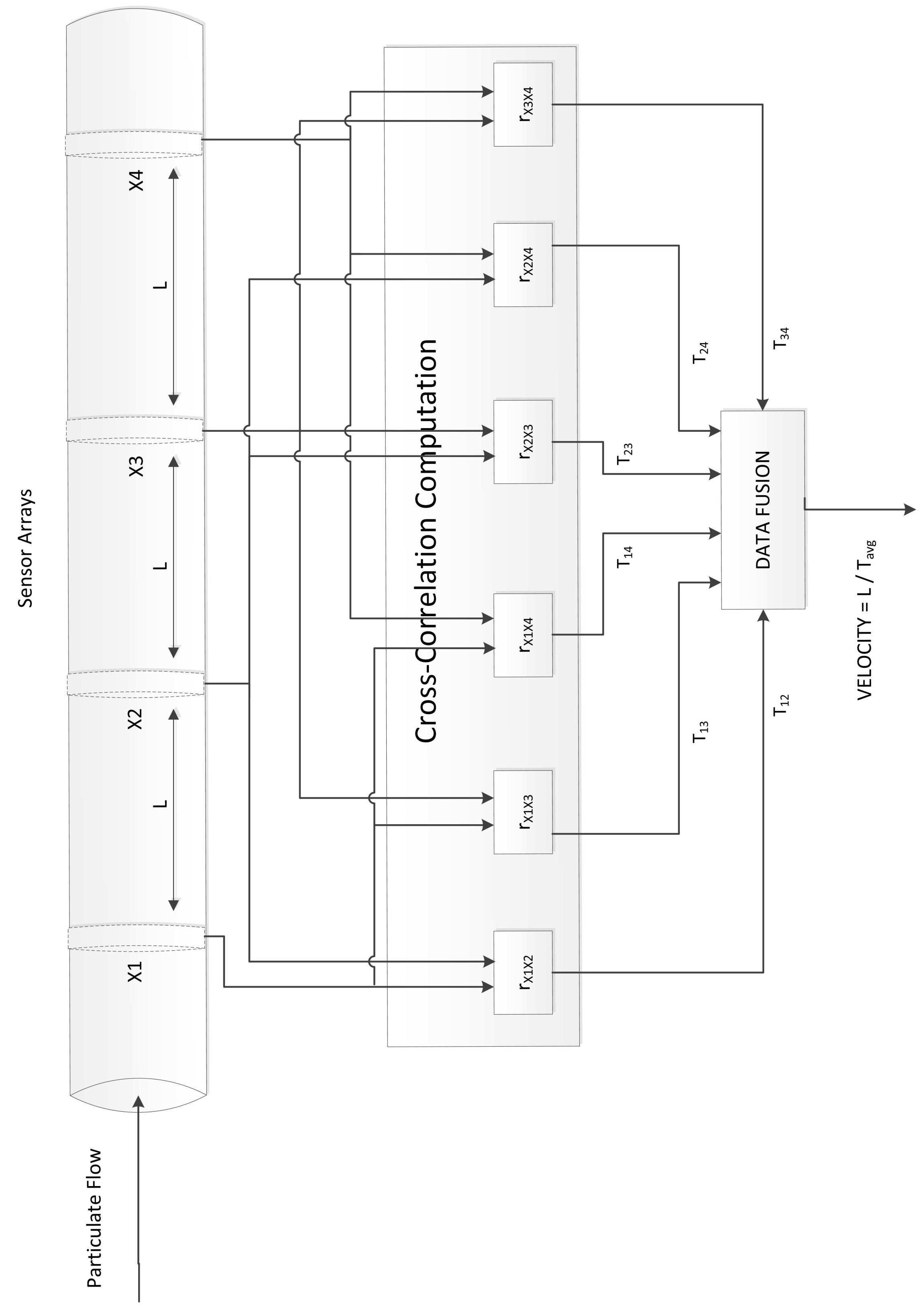


Figure 6.3: 4 channel velocimetry system.

is used to give a robust and reliable solution for calculating the average time of particulates flowing from one sensor to the another. The data fusion block uses simple algorithms such as simple and weight moving average to calculate the time average of the multiple measurement results [126]. Finally, the average velocity of flowing particulates is calculated by performing a division operation between the fixed length ($L$) of equally spaced sensors and calculated average time ($T_{avg}$).

A prototype system based on the Xilinx Spartan 6 family is shown in figure 6.4. The prototype system shows an overview of a 4-channel velocimetry system architecture on an FPGA. This prototype system converts analogue signals to digital by using an AD7476 [166] 12 bits A/D converter. Digital signals are stored in the FIFO (first in first out) registers of respective sizes. Initially working of the FIFO registers in respect to incremental correlation is tested by using DAC121S101 [167] 12 bits D/A converter (see figure 6.4). After performing incremental correlation the output is converted to logarithmic number, which is used for normalisation.

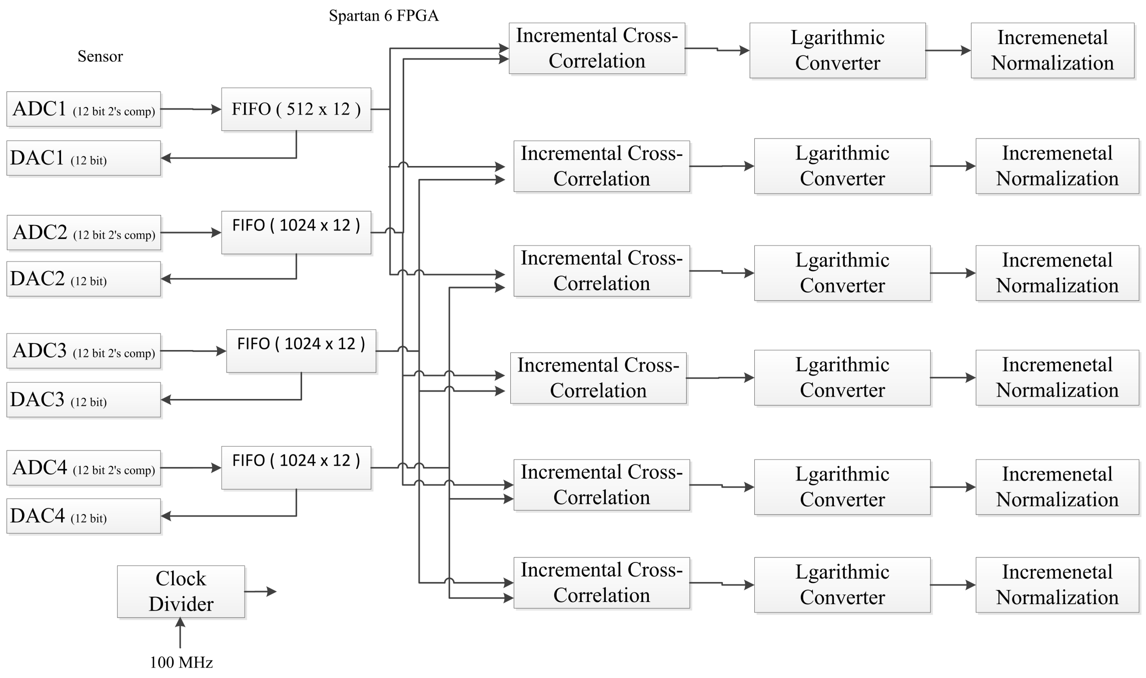


Figure 6.4: Architecture for an FPGA-based 4-channel cross-correlation prototype system.

A generic example of 4 to 8 word depth, upstream and downstream FIFO working is shown in figure 6.5. In this example, the new and old values of the upstream and downstream signals are operated according to their incremental operations (see equations 6.11 and 6.12 ). The FIFO on the right hand-side in figure 6.5 stores a previous value of cross-correlation. The signals $C_{susd}$, $su^2$ and $sd^2$ for normalisation are obtained by adding values stored in the FIFO register to their respective new values and then subtracted from their old values of cross-correlation.

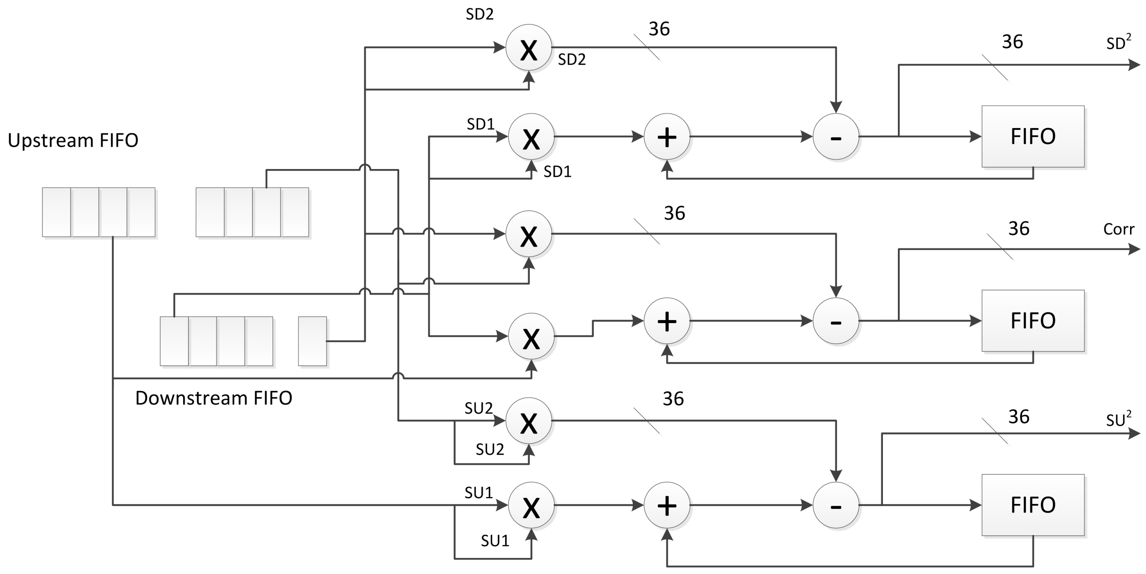


Figure 6.5: Incremental calculations arithmetic unit.

A logarithmic architecture is used for calculating the correlation coefficients as it simplifies the operations of multiplication, division and square root to addition, subtraction and multiplication respectively in the logarithmic domain (see equations 6.13 and 6.14). Figure 6.6 shows (i) the conversion of the signals $su^2$, $sd^2$ and $C_{susd}$ into logarithmic domain and (ii) their normalisation. In Lin2Log conversion, firstly numbers are normalised using leading one detector, where it distinguishes between integer and fractional part of a binary number. The binary number integer part is mapped to an equivalent log value, using integer ROM. A direct ROM conversion is used for the integer part because of having just 36 combinations of output which are 6 bit in resolution. For a binary fractional part, a piecewise linear approximation is used instead of the direct ROM approach to avoid high usage of hardware resources. In the fractional part 16 MSBs are selected and rest of the bits are discarded (out of 36-bit binary value). The selected 16 MSBs are further divided into 8 MSBs and 8 LSBs.

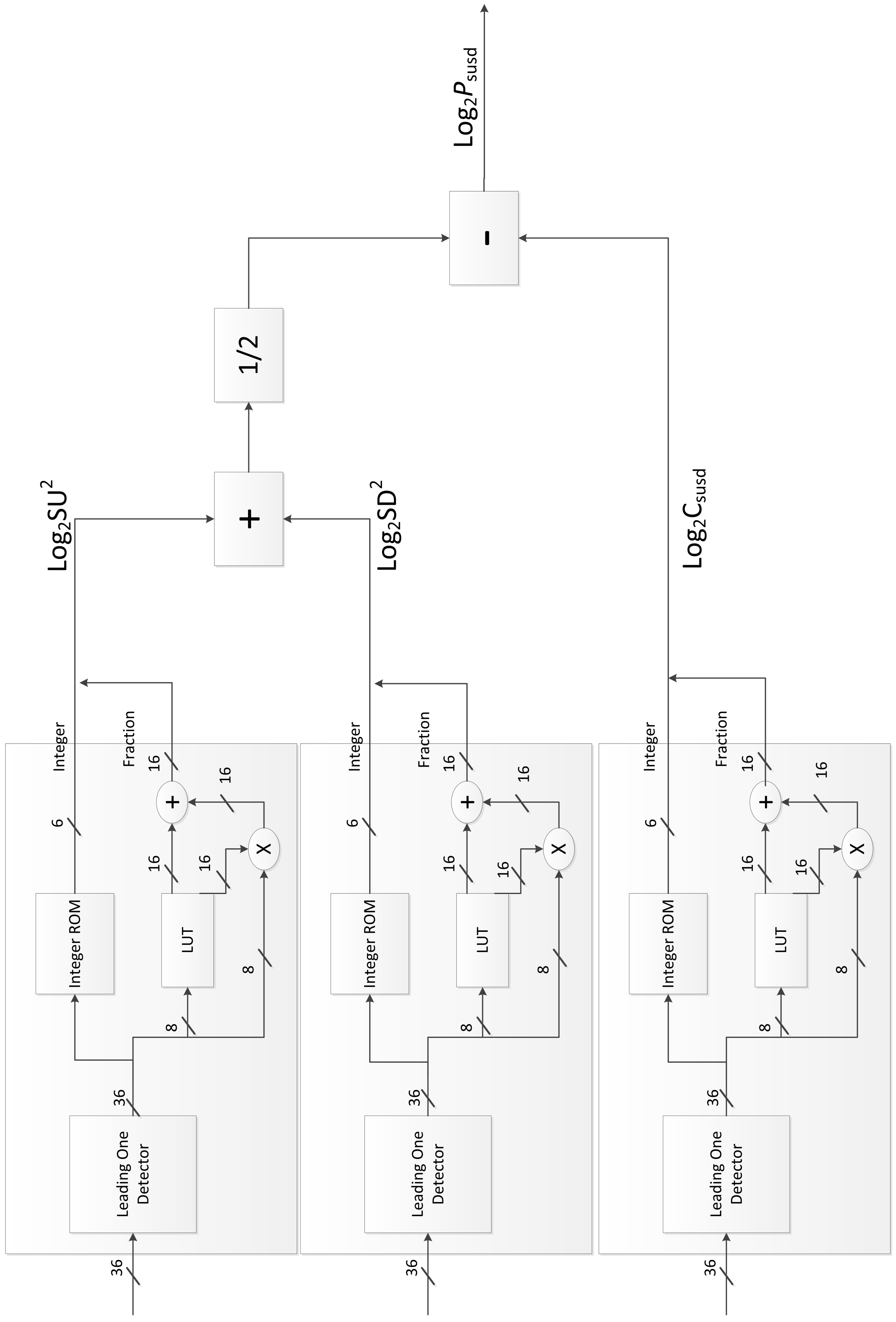

$\text{Log}_2P_{\text{susd}}$
-
1/2
+
$\text{Log}_2\text{SU}^2$
$\text{Log}_2\text{SD}^2$
$\text{Log}_2\text{C}_{\text{susd}}$
Integer
Fraction
16
6
Integer ROM
LUT
X
8
36
Leading One Detector

Figure 6.6: Lin2Log and normalisation circuit.

The 8 MSBs work as an address bit for LUT in piecewise linear approximation, where LUT is storing zeroth and first order coefficient values. The remaining 8 LSBs are multiplied with a first order coefficient value (16 bits in resolution). Multiplication results are added to the zeroth order coefficient value, producing a 16-bit approximated value for a binary fractional part. Integer and fractional parts of the logarithmic number are joined by concatenation. $Log_2 su^2$, $Log_2 sd^2$ and $Log_2 C_{susd}$ obtained by PWL approximation are added, subtracted and multiplied with $\frac{1}{2}$ to calculate the correlation coefficient ($Log_2 \rho_{susd}$).

## 6.7 Results

The cross-correlation was performed on the six combinations of the 4-channel velocimetry system. In the cross-correlation, the time (sample value) was required for finding the velocity of flowing particulates in the pipeline. To calculate the time taken by particulates to flow from one sensor to another, the cross-correlation coefficients of each combination of sensors were plotted. The highest peak in a cross-correlation plot denotes the time (sample) taken by particulates to flow from one sensor (upstream channel) to the other (downstream channel). To verify the cross-correlation operations of each of the six possible combination of sensors, the MATLAB simulations were performed. For testing purposes in MATLAB simulations it is assumed that particulates take 100 samples (time value) to cross from one sensor to another. Figures 6.7, 6.8 and 6.9 show the highest peak (time/sample value) achieved by cross-correlation plots of each of the possible combinations. Once the time (sample) value for each combination was known the next step was to calculate the average time using the simple and weighted moving average. In a simple moving average, all combinations time values are added and divided by the total combination number. The mathematically simple moving average (S.M.A) is represented as

$$S.M.A = \frac{1}{n}\sum_{i=1}^{n} T_i \tag{6.15}$$

where $n$ represents the total number of combinations of sensors and $T_i$ represents the time/sample value of each combination. Figure 6.10a shows the accuracy achieved in a logarithmic approximation based cross-correlation of simple moving average.

Another approach to calculate average time is by performing weighted moving average. In weighted moving average the calculated time for each sensor combination is multiplied by its weight factor and added with other time values. A weighted moving average mathematically can be represented as

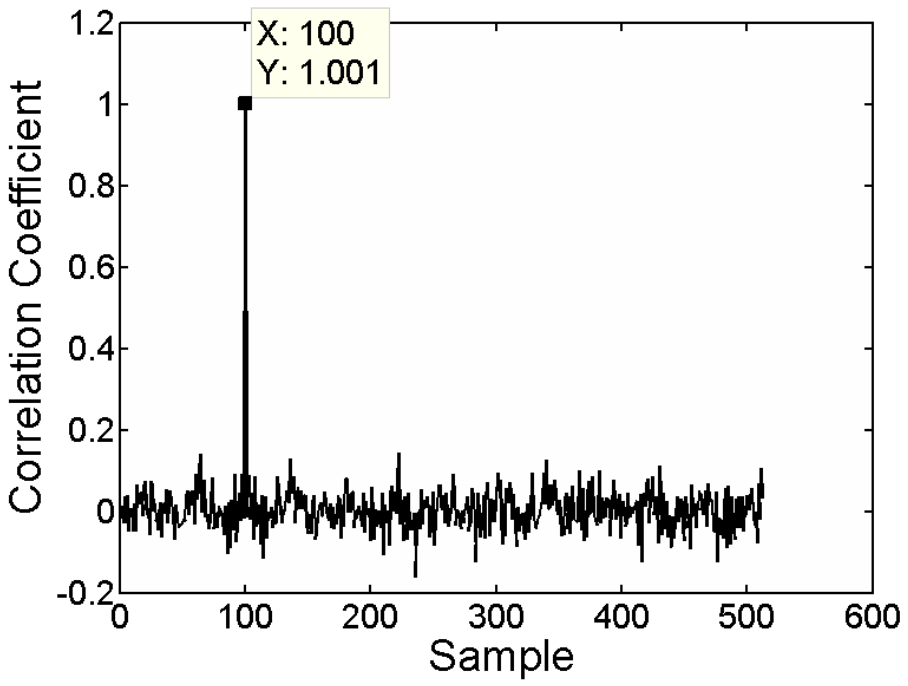
$$W.M.A = \frac{\sum_{i=1}^{n}(n+1-i)T_i}{\sum_{i=1}^{n}(n+1-i)} \tag{6.16}$$

where $n$ represents total number of combinations of sensors and $T_i$ represents the time/sample value of each combination. Figure 6.10b shows the accuracy achieved in logarithmic approximation based cross-correlation of weighted moving average. The advantage of weighted moving average over simple moving average is that it has shorter response time as more recent data has more weighing factor in the averaging process because of which such averaging process are preferred in the time varying process.

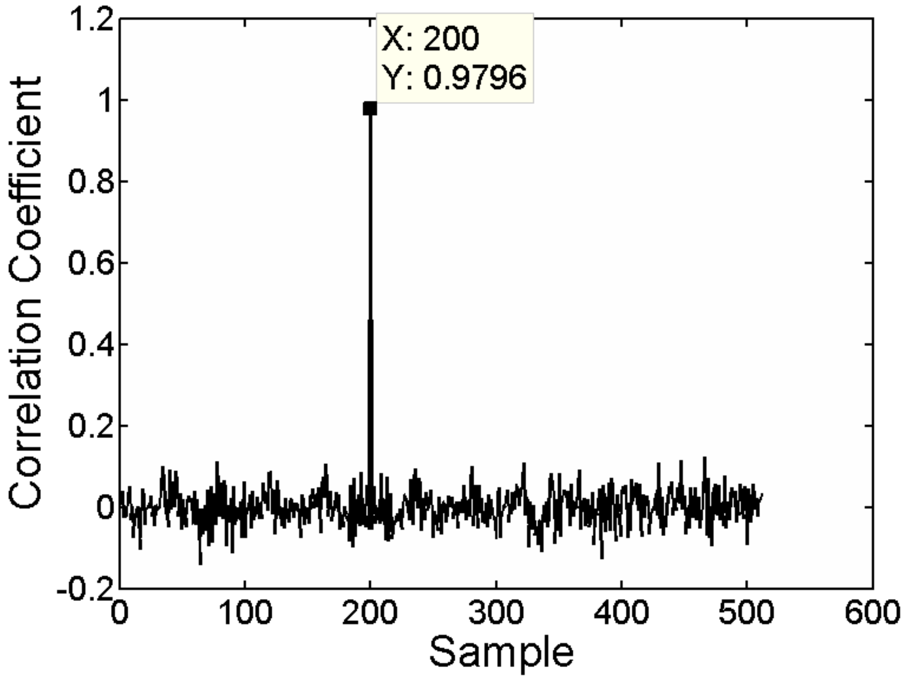
### 6.7.1 Log2Lin Domain

The circuit for cross-correlation can be made simpler and faster by avoiding the computation required for antilogarithm. As per the real-time application, detecting time (sample) without performing antilogarithm on logarithmic cross correlation coefficient value is straightforward. Figures 6.7, 6.8 and 6.9 show the cross-correlation coefficient obtained after performing logarithmic and anti-logarithmic conversions. Figure 6.11 shows the cross-correlation coefficient obtained only after performing logarithmic conversion. The cross-correlation shown in figure 6.11 is of first combination of permutation previously shown in figure 6.7a. Results in figure 6.11 show that time (sample) on the x-axis remains unchanged irrespective of the correlation coefficient value in the y-axis. This observation from plot proves that output on this circuit after logarithmic conversion without performing anti-logarithmic conversion can be used for real-time application. The cross-correlation coefficient plots without using anti-logarithmic conversion for all combinations (by permutation) are shown in appendix E.

The error produced by using 16 bits of fractional bit on a comparison with ideal

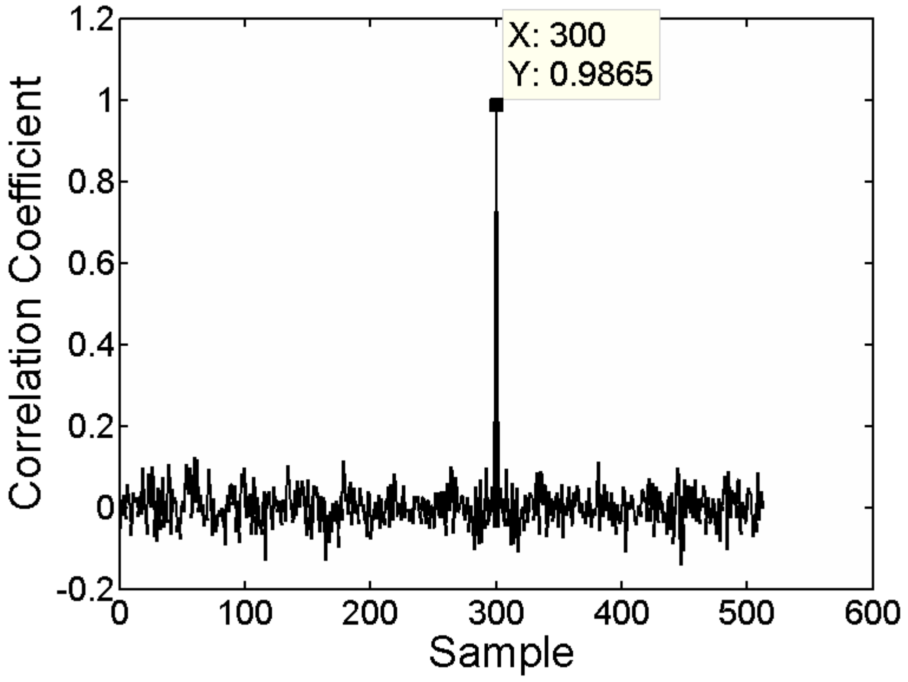


(a)

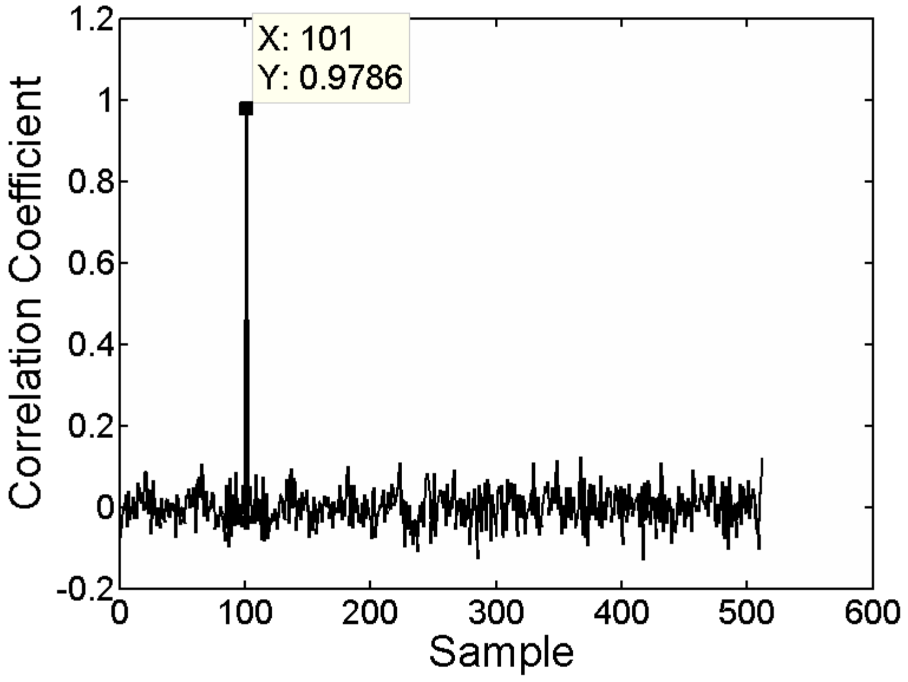


(b)

Figure 6.7: Log Approximation based Cross-correlation of (a)First Combination (b) Second Combination.

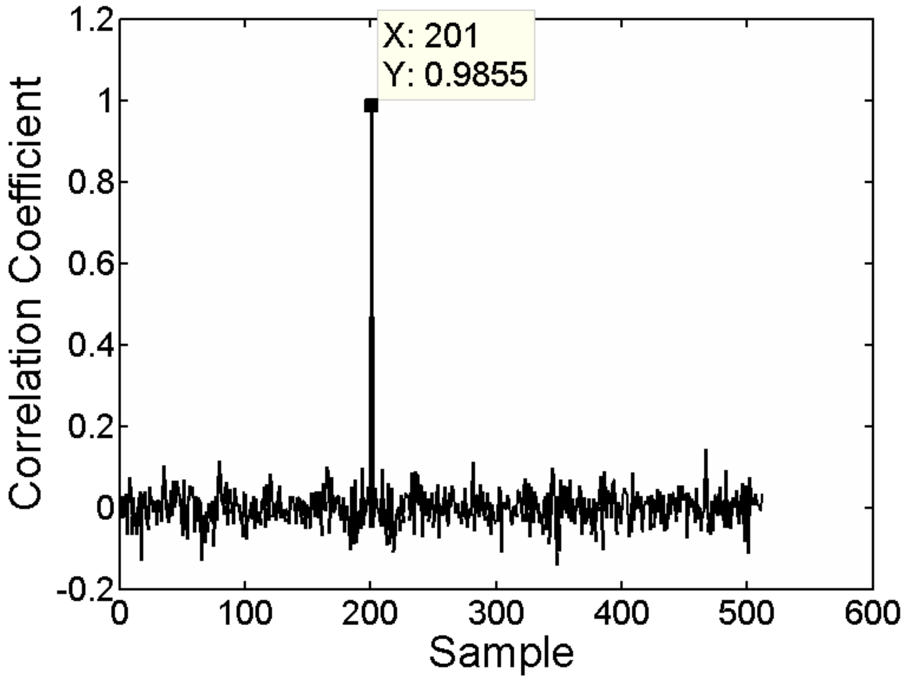


(a)

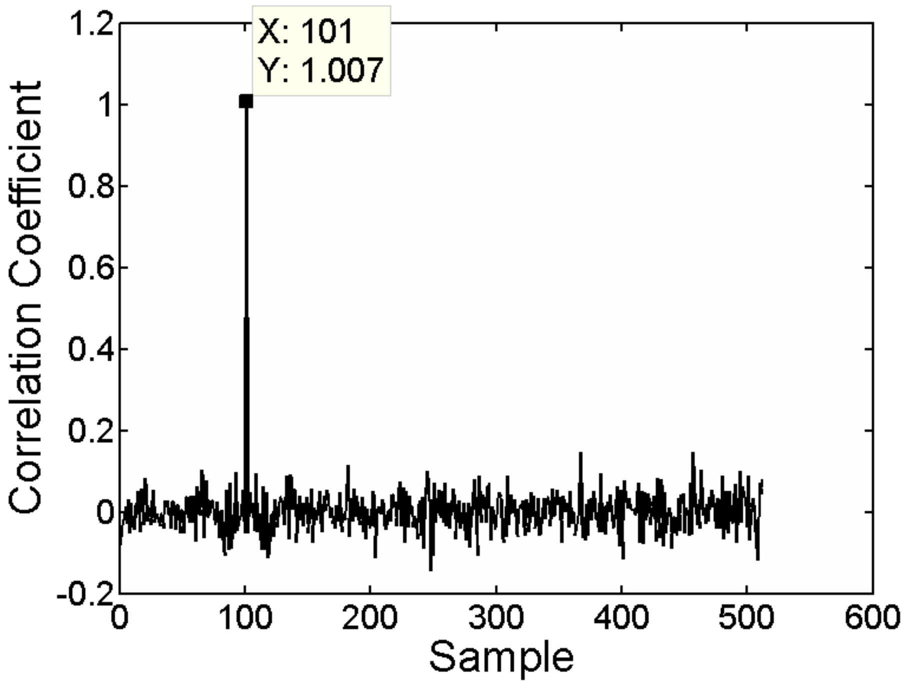


(b)

Figure 6.8: Log Approximation based Cross-correlation of (a)Third Combination (b) Fourth Combination.

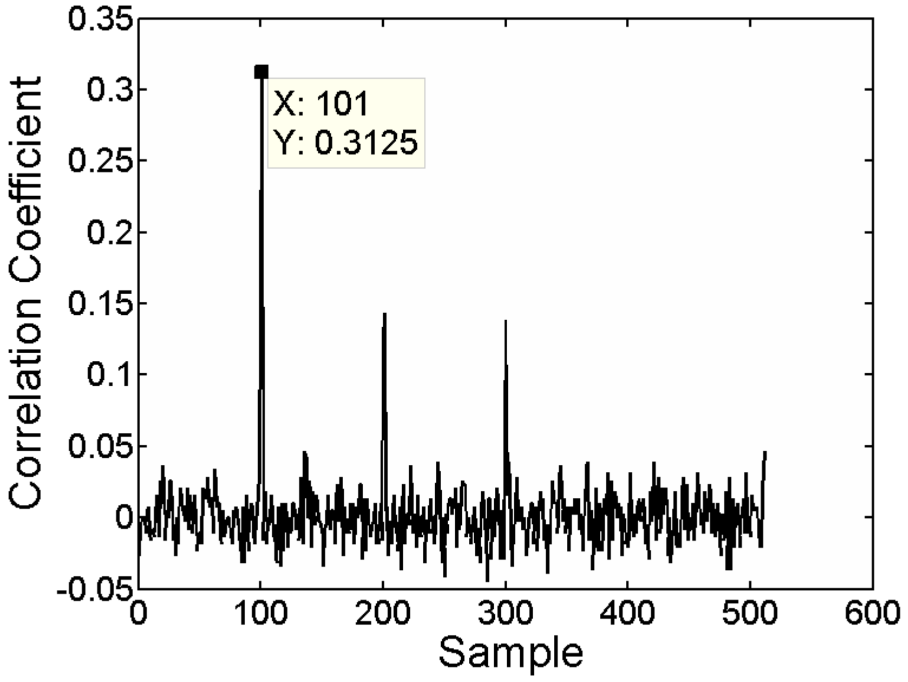


(a)

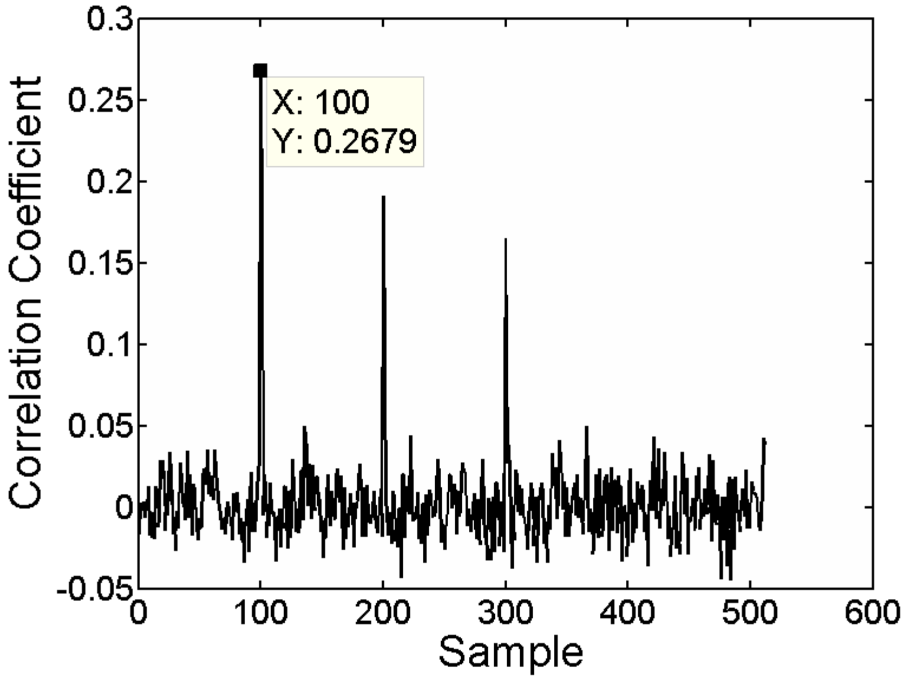


(b)

Figure 6.9: Log Approximation based Cross-correlation of (a)Fifth Combination (b) Sixth Combination.



(a)



(b)

Figure 6.10: Log Approximation based (a)Simple moving average (b)Weighted moving average of six combinations.

cases (MATLAB simulations) has been ignored as it does not affect the computation of the velocity of pneumatically conveyed particles.

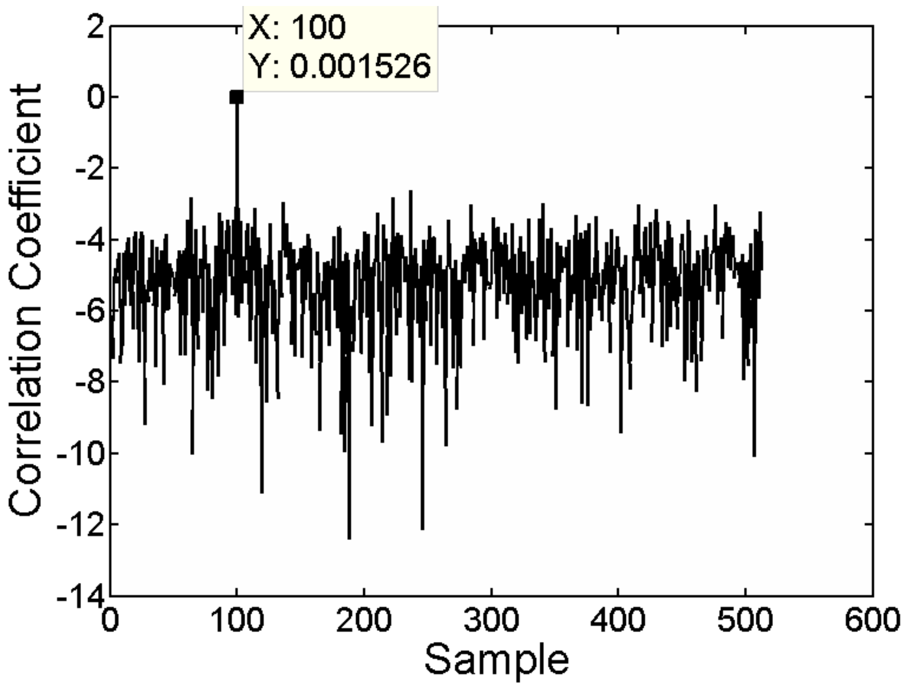


Figure 6.11: Logarithmic cross correlation coefficient value of first combination.

## 6.7.2 FPGA Utilisation

In the prototype device, the sampling rate is set to be 195.31 kHz and the sample resolution is 12 bits. The circuit works on 512 correlation samples resulting in 512 cross-correlations per sample. Architecture described in section 6.6 is first simulated in MATLAB and then written and compiled in VHDL using Xilinx ISE Design Suite 14.6. A top level diagram of 4-channel velocimetry system is shown in figure 6.12. The cross-correlation and normalisation blocks used in 4-channeled velocimetry system are shown in figure 6.13 and 6.14 respectively and their VHDL codes are presented in appendix F.

Due to limited hardware resources on an FPGA Spartan6 board, the prototype system is limited to 512 into 1024 cross-correlations producing results of 512 samples. The prototype device is working at an operating frequency of 100 MHz (oscillator frequency available on a Spartan6 board). The prototype uses 195.31 kHz as the sampling frequency, which is obtained by dividing the operating frequency by the number of required samples i.e.

$$\frac{100 \quad MHz}{512} = 195.31 \quad kHz \tag{6.17}$$

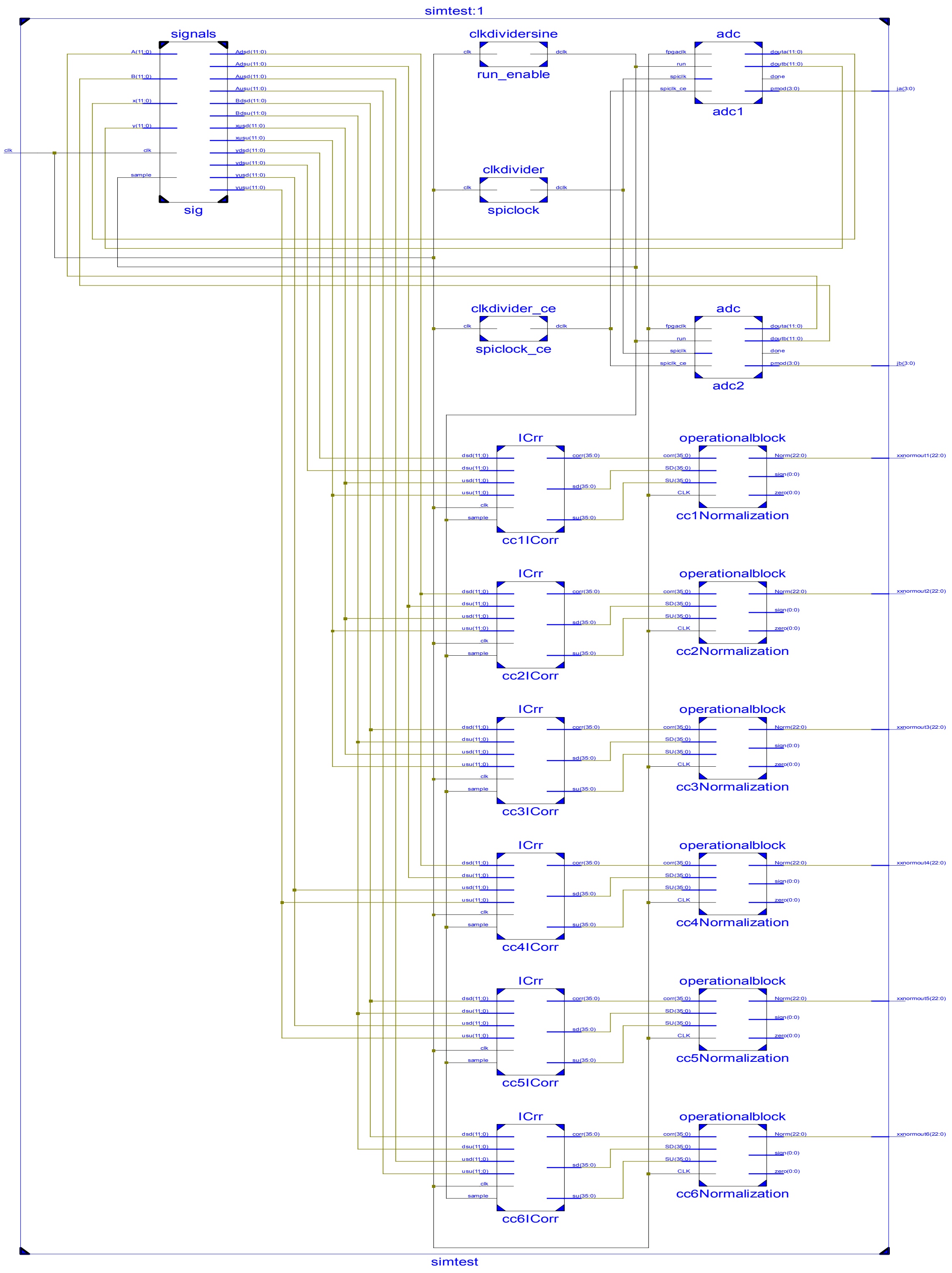


Figure 6.12: Spartan6 based 4-channel velocimetry system.

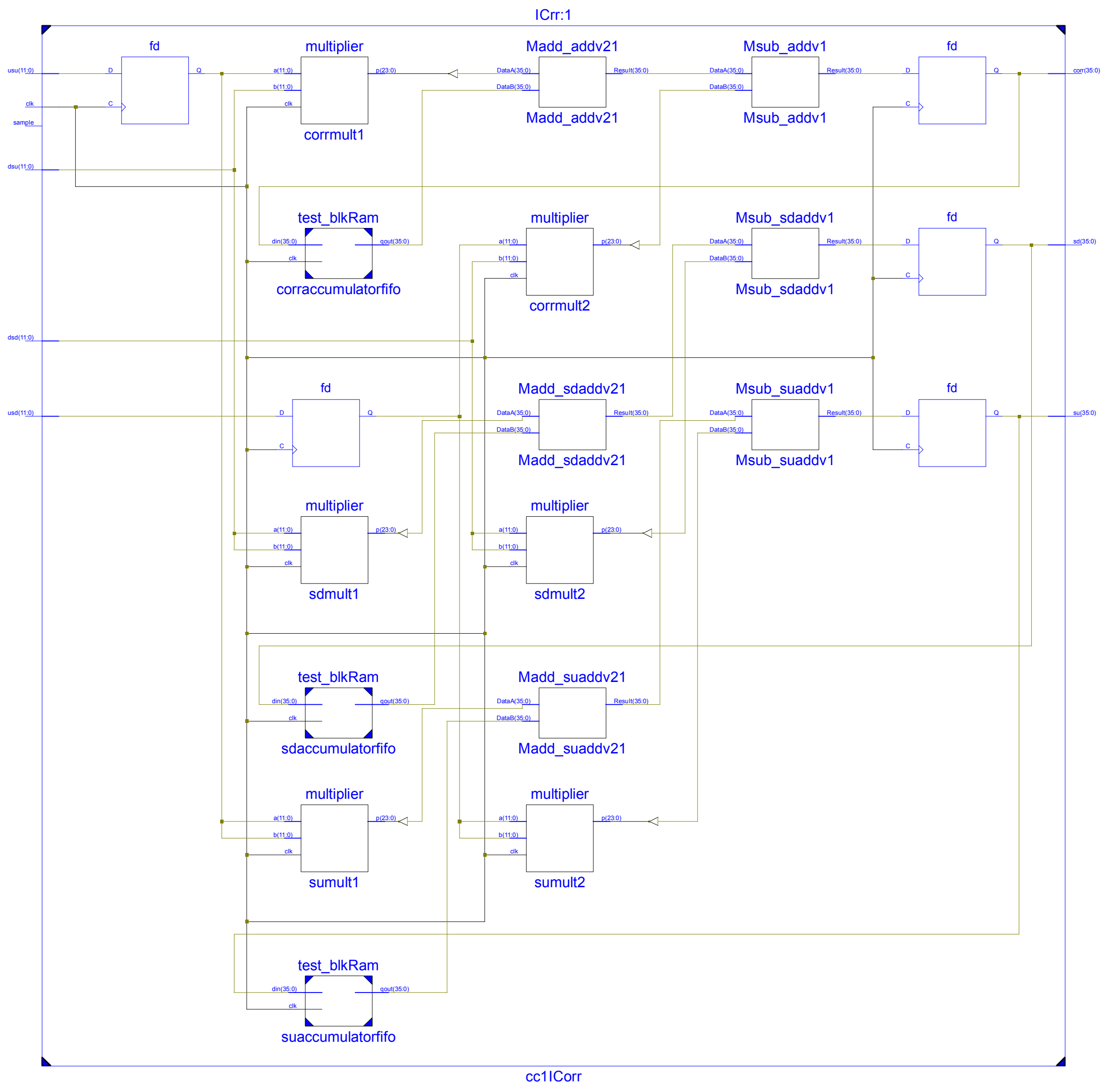


Figure 6.13: Cross-correlation block of velocimtery system.

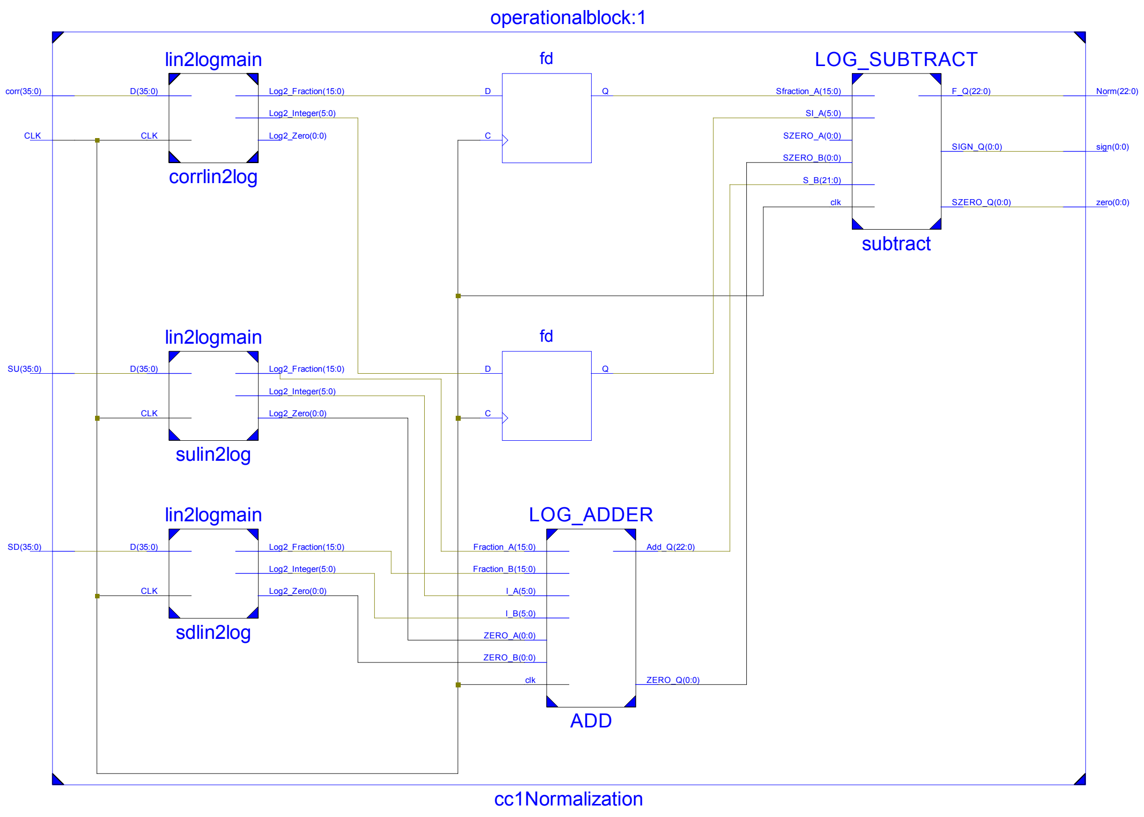


Figure 6.14: Normalisation block of velocimtery system.

$$step\ size(resolution) = \frac{1}{sampling frequency} = \frac{1}{195.31\quad kHz} = 5.12\mu s \tag{6.18}$$

$$range = step\ size \times number of samples = 5.12\mu s \times 512 = 2.6ms \tag{6.19}$$

The cross-correlation circuit calculates the delay in time or the highest peak value of flowing particulates (to be used for finding the velocity of particulate see sections 6.5 and 6.7). The cross-correlation circuit uses 5.12 $\mu s$ step size / resolution (see equation 6.18) to calculate the highest peak value of flowing particulates in the range of 0 to 2.6 $ms$ (see equations 6.19). The prototype system can work with a higher operating frequency in a comparison to the 100 MHz (used in the prototype circuit). However, there is a trade off between the range of calculating the peak value and operating frequency (as shown in equation 6.17, 6.18 and 6.19) due to which 100 MHz is chosen as the operating frequency of the prototype system.

The prototype system has a worst case delay of 13.07 $ns$. However, further analysis of system latency has been ignored in this research as the system uses an incremental correlation algorithm where with every new sample value a new result is outputted (see section 6.2.3). So, the time taken by the complete circuit (or the frequency) to output the new result is directly dependent on the sampling frequency of circuit, which is 195.31 kHz in this case. Hence after the initial set up, the prototype system outputs the new results with a latency of 5.12 $\mu s$.

Initially the experiment was performed on a 2-channel signal without using the anti-logarithm circuit. Table 6.1 shows hardware resources consumption. In Spartan6, due to the new architecture, the complete slices cannot be used as LUT. Due to this, when a 4-channel signal is implemented on Spartan6 on the same device as of 2-channel signal, the slice LUTs and DSP48A1s exceeded the number of hardware resources available on the chip shown in table 6.2. The complete circuit of the 4-channel signal is implemented on a Spartan6 xc6slx45csg324-2 device, where slice LUTs and DSP48A1s are completely implemented on the chip. Hardware resources used by 4-channel signal architecture on the Spartan6 are shown in table 6.3. From table 6.1 to table 6.3 it is also observed that the number of hardware resources such as slice registers, LUTs and DSP48A1s have almost increased by six times. This holds true for six combinations of cross-correlation shown in figure 6.3.

Table 6.1: FPGA implementation of 2-channel signal

| Device | Slice Register | Slice LUTs | Block RAM | DSP48A1s |
|---|---|---|---|---|
| | Used/Available | Used/Available | Used/Available | Used/Available |
| XC6SLX**16** | 789 (4%) | 2233 (24%) | 6 (19%) | 9 (28 %) |
| csg324-2 | 18224 | 9112 | 32 | 32 |

Table 6.2: FPGA implementation of 4-channel signal with exceeded hardware resources in detail

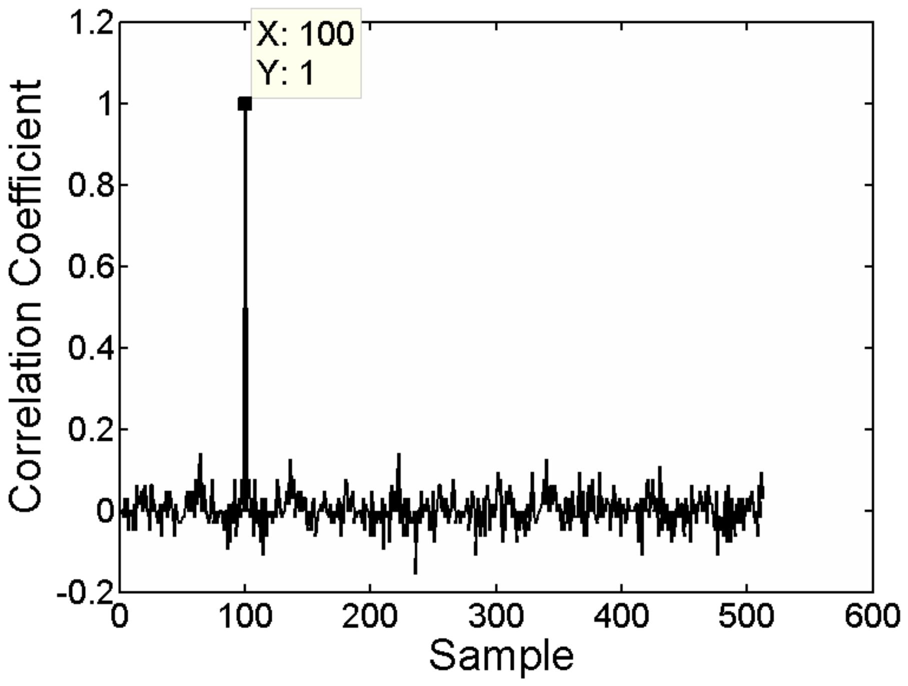
| Device | Slice Register | Slice LUTs | Block RAM | DSP48A1s |
|---|---|---|---|---|
| | Used/Available | Used/Available | Used/Available | Used/Available |
| XC6SLX**16** | 4157 (22%) | 11879 (130%) | 27 (84%) | 54 (168 %) |
| csg324-2 | 18224 | 9112 | 32 | 32 |

Table 6.3: FPGA implementation of 4-channel signal

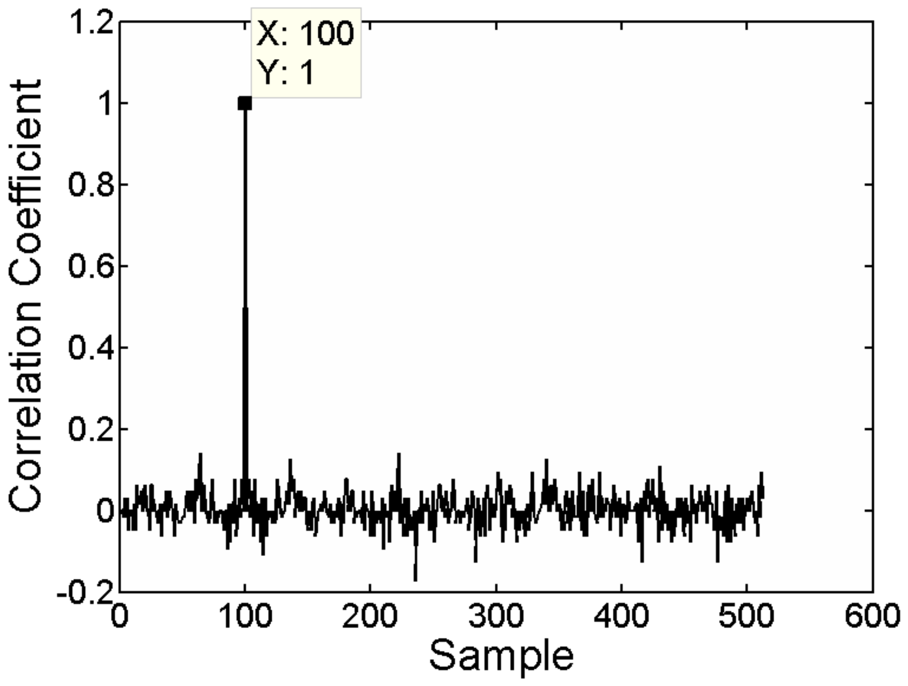
| Device | Slice Register | Slice LUTs | Block RAM | DSP48A1s |
|---|---|---|---|---|
| | Used/Available | Used/Available | Used/Available | Used/Available |
| XC6SLX**45** | 4336 (7%) | 12082 (44%) | 27 (23%) | 54 (93 %) |
| csg324-2 | 54576 | 27288 | 116 | 58 |

### 6.7.3 Piecewise Linear Area Optimisation

The results obtained in section 6.7.1 and 6.7.2 are computed when the fractional bit of log domain is 16 bits in piecewise linear approximation. Experiments have been performed on reducing the resolution of fractional bits. The graphs shown in figures 6.15 and 6.16 show the effect of reducing bits on the first combination in the velocimetry system. In figures 6.15a and 6.15b the resolution of fractional bits (of log domain) used for piecewise linear approximation is reduced from 16 bits to 12 bits and 10 bits respectively. When figure 6.15 is compared with figure 6.7, the accuracy of cross-correlation coefficient reduces (i.e. the magnitude in y-axis) but the value (x-axis value) at which the highest peak occurs remains the same. Similar results are also achieved when the fraction bits (of log domain) are further reduced to 8 and 6 bits in figures 6.16a and 6.16b respectively. Analysis on accuracies reduced when fraction bit resolution is decreased from 16 bits to 12 bits, 10 bits, 8 bits and 6 bits respectively has been ignored due to no change in highest peak value in the x-axis of the cross-correlation coefficient plot. In this research, only the time (sample value) of the highest peak in the cross-correlation coefficient plot is required to calculate the velocity of flowing particulates (see section 6.5), so the reduction in fractional bits of log domain (shown above) can be used to optimise the area of the circuit effectively.

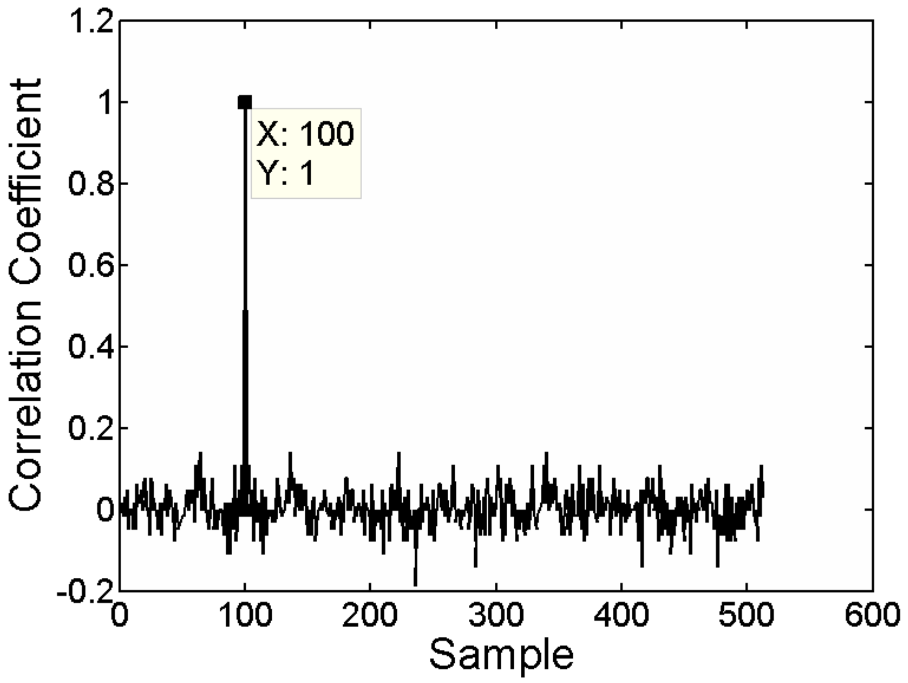


(a)

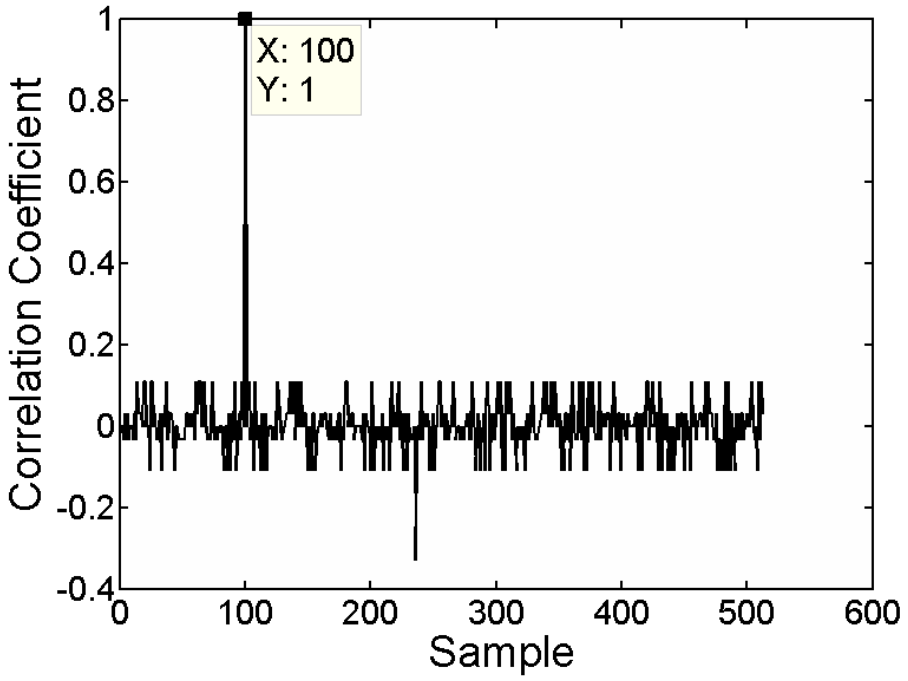


(b)

Figure 6.15: Approximation of first combination when fraction bits are reduced to (a)12 bits (b)10 bits.



(a)



(b)

Figure 6.16: Approximation of first combination when fraction bits are reduced to (a)8 bits (b)6 bits.

## 6.8 Conclusion

The chapter presented a novel FPGA-based 4-channel correlation velocimetry system and an improved 2-channel correlation velocimetry system. The prototype velocimetry system is used for real-time speed measurement of pneumatic particulates flowing through a pipeline. The system is developed using incremental cross-correlation in the time domain instead of using FFT (Fast Fourier Transform) techniques in the frequency domain for the calculation of a continuous stream of data from multiple electrostatic sensors located in a pipeline. The system operates at a higher sampling frequency than previously published work [125–129] and outputs the new result after every new sample it receives. This chapter provides the results of implementing the circuit on an FPGA device and shows the reduction of bits in fractional bits (used for a linear to logarithmic converter). To simplify the computations in cross-correlation the velocimetry system uses LNS. The LNS in the velocimetry system uses an algorithm of piecewise linear approximation to approximate the logarithmic values.

The novel FPGA-based 4-channel correlation velocimetry system is implemented on a Xilinx Spartan6 device. The velocimetry system is working at a sampling frequency of 195.31 kHz and sample resolution of 12 bits. The circuit calculates a delay in a range of 0 to 2.6 $ms$ with a resolution / step size of 5.12 $\mu s$ in cross-correlation plots. The prototype device uses just 6 BRAM and 9 multipliers on an XC6SLX16 device for an improved 2-channel velocimetry system. The 4-channel velocimetry system on an XC6SLX45 device uses 27 BRAM and 54 multipliers for six combinations of cross-correlation.

# Chapter 7

# Conclusions and Future Work

## 7.1 Summary of Research

The thesis has presented two novel algorithms [10,11] for converting a fixed / floating point number to a binary logarithmic number and an application for finding the velocity of pneumatic particles flowing through a pipeline. The thesis has shown the implementation of algorithms and application on new and old families of FPGA devices and compares them with recently published work.

Chapter 1 introduced, in general terms, the concept of LSP (logarithmic signal processing) with its architectures. As stated in Chapter 1, the advantage of LNS (logarithmic number system) is the reduction of multiplication and division operations to addition and subtraction operations. Due to non-linearity, the operations of addition and subtraction (in linear domain) are avoided in the logarithmic domain.

All the circuit designs proposed in this thesis have been implemented using FPGA technology, for the reasons outlined in Chapter 2. FPGA technology is compared with other existing technology and the arithmetic technology used in implementing the proposed circuits is explained. The number systems implemented on FPGA devices are mentioned and compared in this chapter.

A literature review on the popular algorithms for leading one detector and logarithmic converters is presented in Chapter 3. The MATLAB simulation graphs are used for understanding and comparing their accuracy.

K.E. Larson's algorithm [12] first mentioned in Chapter 3 (section 3.3.5) is investigated and a novel algorithm [10] is introduced in Chapter 4. The proposed novel algorithm [10] is analysed and simulated using MATLAB software. For hardware implementation, the novel algorithm is implemented onto newer and old families of FPGA devices. An analysis of different configurations of data bits, used for addressing and interpolation of the PWL (Piece-wise Linear) approximation are shown. A detailed overview of the resolution of coefficient bits stored in memory to perform PWL approximation is also provided in this chapter. The numerical data on precision achieved is compared with recent published papers.

Chapter 5 proposes further improvement to the novel algorithm [10] first proposed in Chapter 4. The improved algorithm [11] reduces the size of memory used for storing the coefficient values. The size of memory is reduced by exploiting the properties of symmetry in the error curve. The chapter also explains how the reduction of memory in hardware is achieved. The proposed algorithm is implemented on FPGA devices and results are compared with the Chapter 4 results and recently published work.

A prototype device for calculating the velocity of pneumatic particles flowing through a pipeline is presented in Chapter 6. The system uses incremental correlation algorithms for producing new results at every new sample. The chapter provides an overview on the recent work done using Kent's method for finding the velocity of biomass and coal particles in a pipeline. The chapter shows no noticeable effect in finding velocity by reducing the number of bits in the fractional bits of a Lin2Log converter. The system is tested using constant distance between electrodes, embedded in a pipe wall and the results are plotted in MATLAB.

## 7.2 Research Novel Claims

The research presented in this thesis claims

- to develop a two-stage novel Lin2Log converter [10] for converting a fixed / floating point number into a logarithmic number. The converter achieves a precision of single floating point number (IEEE 754). The novel method [10] shows empirical data for the accuracy of the converter using a number of different LUT configurations. The novel method [10] also shows the use of less than 20 kbits of ROM and maximum of three multipliers when implemented on new and old families of FPGA devices.

• to further improvement on previously proposed algorithm [10]. The new proposed algorithm [11] exploits the property of symmetry in the first-stage approximated error curve. By using the improved proposed algorithm [11] a 32% reduction in LUT of second-stage approximation is shown. In the hardware implementation on new and old families of FPGA devices, the novel algorithm [11] shows a reduction of BRAM by 50%.

• to propose a novel 4-channel velocimetry system to calculate the velocity of pneumatic particles flowing through a pipeline. The velocimetry system uses an incremental correlation algorithm, making it output the new result with every new sample value. The proposed velocimtery system reduces the complete circuit area by not performing anti-logarithmic operations on the cross-correlation coefficients plots and by reducing the resolution of fractional bits in PWL logarithmic approximation.

## 7.3 Future Research Directions

The novel technique [10] presented in Chapter 4 is suitable for implementation on the new generation of reconfigurable fabrics that now have hardware multipliers together and small blocks of distributed memory on the same chip. The technique explained in this chapter can also be used for a Log2Lin (or $2^{0.F}$) converter. It can also be applied to the architecture of continuous real-time applications such as correlation in instrumentation and communication LTE (Long Term Evolution) decoding of OFDM (Orthogonal Frequency Division Multiplex). In both these applications, complex computations are reduced to simple additions and subtractions, which make computations fast while maintaining sufficient accuracy.

The novel algorithm [11] presented in Chapter 5 can be adapted easily to applications needing higher accuracies. This method is even more effective on newer Xilinx devices which have increased distributed memory capacities over earlier generations. Future work will investigate the use of these methods at higher precision and investigate if they can be applied when high order PWP approximations are used. The proposed algorithm can also be applied to other functions such as $2^x$, $\frac{1}{x}$, $\sqrt{x}$, $\sin(x)$ and $\frac{1}{\sqrt{x}}$.

For the prototype device presented in Chapter 6, it will be interesting to investigate the area optimisation of circuit implemented on an FPGA by using multiplexers in the circuit and calculating the latency caused by them. The FPGA prototype device can be tested with different velocimetry systems using sensors such as circular, radiometric, optical and combinations of them.

# Bibliography

[1] F. Sheikh, S. Mathew, M. Anders, H. Kaul, S. Hsu, A. Agarwal, R. Krishnamurthy, and S. Borkar, "A 2.05gvertices/s 151mw lighting accelerator for 3D graphics vertex and pixel shading in 32nm CMOS," *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International*, pp. 184–186, Feb 2012.

[2] V. Paliouras, "Optimization of LNS operations for embedded signal processing applications," *Circuits and Systems, 2002. ISCAS 2002. IEEE International Symposium on*, vol. 2, pp. II–744–II–747, 2002.

[3] R. Ismail, R. Hussin, and S. Murad, "Interpolator algorithms for approximating the lns addition and subtraction: Design and analysis," *Circuits and Systems (ICCAS), 2012 IEEE International Conference on*, pp. 174–179, Oct 2012.

[4] C. Chen and C. H. Yang, "Pipelined computation of LNS addition/subtraction with very small lookup tables," *Computer Design: VLSI in Computers and Processors, 1998. ICCD '98. Proceedings. International Conference on*, pp. 292–297, Oct 1998.

[5] S. Melnikoff, S. Quigley, and M. Russell, "Speech recognition on an FPGA using discrete and continuous hidden Markov models," *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, ser. Lecture Notes in Computer Science, vol. 2438, M. Glesner, P. Zipf, and M. Renovell, Eds., pp. 202–211, 2002.

[6] H. Li, G. Jullien, V. Dimitrov, M. Ahmadi, and W. Miller, “A 2-digit multidimensional logarithmic number system filterbank for a digital hearing aid architecture,” *Circuits and Systems, 2002. ISCAS 2002. IEEE International Symposium on*, vol. 2, pp. II–760–II–763, 2002.

[7] P. Lee, “A VLSI Implementation of a Digital Hybrid LNS Neuron,” *Integrated Circuits, 2007. ISIC ’07. International Symposium on*, pp. 9–12, Sept 2007.

[8] M. Arnold, “Reduced power consumption for MPEG decoding with LNS,” *Application-Specific Systems, Architectures and Processors, 2002. Proceedings. The IEEE International Conference on*, pp. 65–75, 2002.

[9] “IEEE standard for floating-point arithmetic,” *IEEE Std 754-2008*, pp. 1–70, Aug 2008.

[10] M. Chaudhary and P. Lee, “Two-stage logarithmic converter with reduced memory requirements,” *Computers Digital Techniques, IET*, vol. 8, no. 1, pp. 23–29, January 2014.

[11] M. Chaudhary and P. Lee, “An improved two-step binary logarithmic converter for fpgas,” *Circuits and Systems II: Express Briefs, IEEE Transactions on*, vol. 62, no. 5, pp. 476–480, May 2015.

[12] K. Larson, “Floating point to logarithm converter,” U.S. Patent 5,365,465. Nov. 15 1994.

[13] I. Kuon and J. Rose, “Measuring the gap between FPGAs and ASICs,” *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays*, ser. FPGA ’06, pp. 21–30. New York, NY, USA: ACM, 2006.

[14] “Tms320c6455 fixed-point digital signal processor,” Texas Instruments, Texas Instruments, Post Office Box 655303, Dallas, Texas 75265, U.S.A.

[15] D. G. Bailey, *Image Processing*, pp. 1–19. John Wiley & Sons (Asia) Pte Ltd, 2011.

[16] D. Weeren, “Minimig.” [Online]. Available: http://www.minimig.net/ Last visited 2014-08-10.

[17] R. H. Klenke, “Experiences using the Xilinx Microblaze softcore processor and uCLinux in computer engineering capstone senior design projects,” *Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education*, ser. MSE ’07, pp. 123–124. Washington, DC, USA: IEEE Computer Society, 2007.

[18] J. Williams, “Microblaze uCLinux project.” [Online]. Available: http://osdir.com/ml/linux.uclinux.microblaze/2005-10/msg00101.html Last visited 2014-08-15.

[19] B. Nelson, “The BYU Linux on FPGA project.” [Online]. Available: http://splish.ee.byu.edu/projects/LinuxFPGA/ Last visited 2014-08-16.

[20] “Cyclone FPGA Family Data Sheet,” Altera Corporation, 101 Innovation Drive, San Jose, CA 95134, U.S.A.

[21] “Spartan-3 FPGA family data sheet,” Xilinx, Inc., San Jose, CA, USA.

[22] “Spartan-3 Starter Kit Board User Guide,” Digilent, Inc, 1300 NE. Henley Court Pullman, WA 99163, U.S.A.

[23] “Spartan-6 FPGA DSP48A1 slice user guide,” Xilinx, Inc., San Jose, CA, USA.

[24] “Spartan-6 FPGA family data sheet,” Xilinx, Inc., San Jose, CA, USA.

[25] “Nexys3 Board Reference Manual,” Digilent, Inc, 1300 NE. Henley Court Pullman, WA 99163, U.S.A.

[26] P. Belanovic and M. Leeser, “A library of parameterized floating-point modules and their use,” *Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, ser. FPL ’02, pp. 657–666. London, UK, UK: Springer-Verlag, 2002.

[27] L. Louca, T. Cook, and W. Johnson, "Implementation of IEEE single precision floating point addition and multiplication on FPGAs," *FPGAs for Custom Computing Machines, 1996. Proceedings. IEEE Symposium on*, pp. 107–116, Apr 1996.

[28] B. Fagin and C. Renard, "Field programmable gate arrays and floating point arithmetic," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 2, no. 3, pp. 365–367, Sept 1994.

[29] K. Underwood, "FPGAs vs. CPUs: Trends in Peak Floating-point Performance," *Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, ser. FPGA '04, pp. 171–180. New York, NY, USA: ACM, 2004.

[30] N. Shirazi, A. Walters, and P. Athanas, "Quantitative analysis of floating point arithmetic on FPGA based custom computing machines," *FPGAs for Custom Computing Machines, 1995. Proceedings. IEEE Symposium on*, pp. 155–162, Apr 1995.

[31] K. Hemmert and K. Underwood, "An analysis of the double-precision floating-point FFT on FPGAs," *Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE Symposium on*, pp. 171–180, April 2005.

[32] S. Akhter and S. Chaturvedi, "HDL based implementation of N x00D7;N bit-serial multiplier," *Signal Processing and Integrated Networks (SPIN), 2014 International Conference on*, pp. 470–474, Feb 2014.

[33] K. Y. Chang, D. won Hong, and H.-S. Cho, "Low complexity bit-parallel multiplier for GF(2m) defined by all-one polynomials using redundant representation," *Computers, IEEE Transactions on*, vol. 54, no. 12, pp. 1628–1630, Dec 2005.

[34] M. Arnold, T. Bailey, J. Cowles, and J. Cupal, "Redundant logarithmic number systems," *Computer Arithmetic, 1989., Proceedings of 9th Symposium on*, pp. 144–151, Sep 1989.

[35] M. Arnold, "A pipelined LNS ALU," *VLSI, 2001. Proceedings. IEEE Computer Society Workshop on*, pp. 155–161, May 2001.

[36] M. Arnold and S. Collange, "A dual-purpose real/complex logarithmic number system ALU," *Computer Arithmetic, 2009. ARITH 2009. 19th IEEE Symposium on*, pp. 15–24, June 2009.

[37] T. Stouraitis and C. Chen, "Hybrid signed digit logarithmic number system processor," *Computers and Digital Techniques, IEE Proceedings E*, vol. 140, no. 4, pp. 205–210, Jul 1993.

[38] D. Lewis, "114 MFLOPS logarithmic number system arithmetic unit for DSP applications," *Solid-State Circuits, IEEE Journal of*, vol. 30, no. 12, pp. 1547–1553, Dec 1995.

[39] S. Gammino, L. Torrisi, S. Cavallaro, L. Celona, L. Giuffrida, D. Margarone, D. Mascali, and R. Miracoli, "Recent results of the laser ion source facility at INFN-LNS and applications to nuclear and applied research," *Review of Scientific Instruments*, vol. 81, no. 2, pp. 02A508–02A508–3, Feb 2010.

[40] L. Cahill and G. Deng, "An overview of logarithm-based image processing techniques for biomedical applications," *Digital Signal Processing Proceedings, 1997. DSP 97., 1997 13th International Conference on*, vol. 1, pp. 93–96, Jul 1997.

[41] H. Li, R. Muscedere, G. Jullien, and V. Dimitrv, "The application of 2-D logarithms to low-power hearing-aid processors," *Circuits and Systems, 2002. MWSCAS-2002. The 2002 45th Midwest Symposium on*, vol. 3, pp. III–13–III–16, Aug 2002.

[42] V. Dimitrov, J. Eskritt, L. Imbert, G. Jullien, and W. Miller, "The use of the multi-dimensional logarithmic number system in DSP applications," *Computer Arithmetic, 2001. Proceedings. 15th IEEE Symposium on*, pp. 247–254, 2001.

[43] J. T.J. Sullivan, R.E. Morley and G. Engel, "A VLSI FIR digital signal processor using logarithmic arithmetic," *IEEE Workshop on VLSI Signal Processing*, vol. VLSI Signal Processing-111, IEEE Press, pp. 276–280, 1988.

[44] P. Lee, "Hybrid-logarithmic arithmetic and applications," Ph.D. dissertation, University of Kent, 2011.

[45] M. S. Schmookler and K. Nowka, "Leading zero anticipation and detection-a comparison of methods," *Computer Arithmetic, 2001. Proceedings. 15th IEEE Symposium on*, pp. 7–12, 2001.

[46] V. Oklobdzija, "An implementation algorithm and design of a novel leading zero detector circuit," *Signals, Systems and Computers, 1992. 1992 Conference Record of The Twenty-Sixth Asilomar Conference on*, vol. 1, pp. 391–395, Oct 1992.

[47] V. Oklobdzija, "An algorithmic and novel design of a leading zero detector circuit: comparison with logic synthesis," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 2, no. 1, pp. 124–128, March 1994.

[48] V. Oklobdzija, H. Suzuki, H. Morinaka, H. Makino, Y. Nakase, K. Mashiko, and T. Sumi, "Comments on ldquo; leading-zero anticipatory logic for high-speed floating point addition rdquo;," *Solid-State Circuits, IEEE Journal of*, vol. 32, no. 2, pp. 292–292, Feb 1997.

[49] M. Schmookler and D. Mikan, "Two state leading zero/one anticipator (lza)," U.S. Patent 5,493,520. Feb. 20 1996.

[50] H. Suzuki, H. Morinaka, H. Makino, Y. Nakase, K. Mashiko, and T. Sumi, "Leading-zero anticipatory logic for high-speed floating point addition," *Solid-State Circuits, IEEE Journal of*, vol. 31, no. 8, pp. 1157–1164, Aug 1996.

[51] J. Bruguera and T. Lang, "Leading-one prediction scheme for latency improvement in single datapath floating-point adders," *Computer Design: VLSI in Computers and Processors, 1998. ICCD '98. Proceedings. International Conference on*, pp. 298–305, Oct 1998.

[52] J. Bruguera and T. Lang, “Leading-one prediction with concurrent position correction,” *Computers, IEEE Transactions on*, vol. 48, no. 10, pp. 1083–1097, Oct 1999.

[53] V. Oklobdzija, “Algorithmic design of a hierarchical and modular leading zero detector circuit,” *Electronics Letters*, vol. 29, no. 3, pp. 283–284, Feb 1993.

[54] P. Lee and A. Sartori, “Modular leading one detector for logarithmic encoder,” *Electronics Letters*, vol. 34, no. 8, pp. 727–728, Apr 1998.

[55] K. Abed and R. Siferd, “CMOS VLSI implementation of 16-bit logarithm and anti-logarithm converters,” *Circuits and Systems, 2000. Proceedings of the 43rd IEEE Midwest Symposium on*, vol. 2, pp. 776–779, 2000.

[56] K. Abed and R. Siferd, “CMOS VLSI implementation of a low-power logarithmic converter,” *Computers, IEEE Transactions on*, vol. 52, no. 11, pp. 1421–1433, Nov 2003.

[57] K. Abed and R. Siferd, “VLSI implementation of a low-power antilogarithmic converter,” *Computers, IEEE Transactions on*, vol. 52, no. 9, pp. 1221–1228, Sept 2003.

[58] K. Abed and R. Siferd, “VLSI implementations of low-power leading-one detector circuits,” *SoutheastCon, 2006. Proceedings of the IEEE*, pp. 279–284, March 2006.

[59] J. N. Mitchell, “Computer multiplication and division using binary logarithms,” *Electronic Computers, IRE Transactions on*, vol. EC-11, no. 4, pp. 512–517, Aug 1962.

[60] M. Combet, H. Van Zonneveld, and L. Verbeek, “Computation of the base two logarithm of binary numbers,” *Electronic Computers, IEEE Transactions on*, vol. EC-14, no. 6, pp. 863–867, Dec 1965.

[61] E. L. Hall, D. Lynch, and I. Dwyer, S.J., “Generation of products and quotients using approximate binary logarithms for digital filtering applications,” *Computers, IEEE Transactions on*, vol. C-19, no. 2, pp. 97–105, Feb 1970.

[62] B. Hoefflinger, "Efficient VLSI digital logarithmic codecs," *Electronics Letters*, vol. 27, no. 13, pp. 1132–1134, June 1991.

[63] B. Hoefflinger, M. Selzer, and F. Warkowski, "Digital logarithmic CMOS multiplier for very-high-speed signal processing," *Custom Integrated Circuits Conference, 1991., Proceedings of the IEEE 1991*, pp. 16.7/1–16.7/5, May 1991.

[64] B. Hofflinger, "Circuit arrangement for digital multiplication of integers," U.S. Patent 5,956,264. Sep. 21 1999.

[65] B. Hoefflinger, "Schaltungsanordnung zum digitalen multiplizieren von integer-zahlen," dE Patent App. DE19,924,213,107. Sep. 2 1993.

[66] E. Der, "Schaltungsanordnung zum digitalen multiplizieren von integer-zahlen," dE Patent App. DE19,904,033,507. Jul. 2 1992.

[67] S. SanGregory, C. Brothers, D. Gallagher, and R. Siferd, "A fast, low-power logarithm approximation with CMOS VLSI implementation," *Circuits and Systems, 1999. 42nd Midwest Symposium on*, vol. 1, pp. 388–391, 1999.

[68] T. Brubaker and J. Becker, "Multiplication using logarithms implemented with read-only memory," *Computers, IEEE Transactions on*, vol. C-24, no. 8, pp. 761–765, Aug 1975.

[69] G. Kmetz, "In a digital computation system," U.S. Patent 4,583,180. Apr. 15 1986.

[70] R. Maenner, "A fast integer binary logarithm of large arguments," *Micro, IEEE*, vol. 7, no. 6, pp. 41–45, Dec 1987.

[71] M. Arnold, T. Bailey, and J. Cowles, "Error analysis of the Kmetz/Maenner algorithm," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 33, no. 1-2, pp. 37–53, 2003.

[72] D. Marino, "New algorithms for the approximate evaluation in hardware of binary logarithms and elementary functions," *Computers, IEEE Transactions on*, vol. C-21, no. 12, pp. 1416–1421, Dec 1972.

[73] R. de Mori and R. Cardin, “A new design approach to binary logarithm computation,” *Signal Processing*, vol. 13, no. 2, pp. 177 – 195, 1987.

[74] D. Lewis, “Interleaved memory function interpolators with application to an accurate LNS arithmetic unit,” *Computers, IEEE Transactions on*, vol. 43, no. 8, pp. 974–982, Aug 1994.

[75] G. Knittel, “A fast logarithm converter,” *ASIC Conference and Exhibit, 1994. Proceedings., Seventh Annual IEEE International*, pp. 450–453, Sep 1994.

[76] S. Pan, S. Panchumarthi, R. Srinath, and S. Wang, “Method and system for solving linear systems,” U.S. Patent 6,078,938. Jun. 20 2000.

[77] S. Pan and S. Wang, “Logarithm/inverse-logarithm converter and method of using same,” U.S. Patent 5,642,305. Jun. 24 1997.

[78] S. Pan and S. Wang, “Log converter utilizing offset and method of use thereof,” U.S. Patent 6,065,031. May 16 2000.

[79] S. Pan and S. Wang, “Logarithm/inverse-logarithm converter utilizing second-order term and method of using same,” EP Patent App. EP19,960,902,604. Sep. 1 1999.

[80] S. Pan and S. Wang, “Logarithm/inverse-logarithm converter utilizing second-order term and method of using same,” U.S. Patent 5,703,801. Dec. 30 1997.

[81] S. Pan and S. Wang, “Logarithm/inverse-logarithm converter utilizing second-order term and method of using same,” WO Patent App. PCT/US1996/000,147. Aug. 8 1996.

[82] S. Pan and S. Wang, “Logarithm/inverse-logarithm converter and method of using same,” U.S. Patent 5,941,939. Aug. 24 1999.

[83] S. Pan and S. Wang, “Method and system for performing an l2 norm operation,” U.S. Patent 5,936,871. Aug. 10 1999.

[84] S. Pan, S. Wang, B. Sigmon, S. Ma, K. Laird, and J. Toler, “Apparatus using a logarithm based processor,” U.S. Patent 5,961,579. Oct. 5 1999.

[85] P. Wei and W. T, “Procede et dispositif concernant une operation en norme l¿2?” WO Patent App. PCT/US1996/013,068. Mar. 6 1997.

[86] L. Bangqiang, H. Ling, and Y. Xiao, “Base-n logarithm implementation on fpga for the data with random decimal point positions,” *Signal Processing and its Applications (CSPA), 2013 IEEE 9th International Colloquium on*, pp. 17–20, March 2013.

[87] A. Vazquez, J. Villalba, and E. Antelo, “Computation of decimal transcendental functions using the cordic algorithm,” *Computer Arithmetic, 2009. ARITH 2009. 19th IEEE Symposium on*, pp. 179–186, June 2009.

[88] D. Chen, L. Han, Y. Choi, and S.-B. Ko, “Improved decimal floating-point logarithmic converter based on selection by rounding,” *Computers, IEEE Transactions on*, vol. 61, no. 5, pp. 607–621, May 2012.

[89] M. S. A. A. M. Mansour, A. M. El-Sawy and A. T. Sayed, “A New Hardware Implementation of Base 2 Logarithm for FPGA,” *International Journal of Signal Processing Systems*, vol. 3, pp. 177–182, December 2015.

[90] D. Kostopoulos, “An algorithm for the computation of binary logarithms,” *Computers, IEEE Transactions on*, vol. 40, no. 11, pp. 1267–1270, Nov 1991.

[91] M. Arnold and M. Winkel, “A single-multiplier quadratic interpolator for LNS arithmetic,” *Computer Design, 2001. ICCD 2001. Proceedings. 2001 International Conference on*, pp. 178–183, 2001.

[92] H. Henkel, “Improved accuracy for the logarithmic number system,” *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. Assp-37, pp. 301–303, Feb 1989.

[93] A. Noetzel, “An interpolating memory unit for function evaluation: analysis and design,” *Computers, IEEE Transactions on*, vol. 38, no. 3, pp. 377–384, Mar 1989.

[94] J. N. Coleman, E. I. Chester, C. Softley, and J. Kadlec, "Arithmetic on the european logarithmic microprocessor," *Computers, IEEE Transactions on*, vol. 49, no. 7, pp. 702–715, Jul 2000.

[95] D. Lewis, "An architecture for addition and subtraction of long word length numbers in the logarithmic number system," *Computers, IEEE Transactions on*, vol. 39, no. 11, pp. 1325–1336, Nov 1990.

[96] L. Yu and D. Lewis, "A 30-b integrated logarithmic number system processor," *Solid-State Circuits, IEEE Journal of*, vol. 26, no. 10, pp. 1433–1440, Oct 1991.

[97] S. C. Huang, L. G. Chen, and T. H. Chen, "A 32-bit logarithmic number system processor," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 14, no. 3, pp. 311–319, 1996.

[98] P. Huang, D. Y. Teng, K. Wahid, and S. B. Ko, "Convergence analysis of jacobi iterative method using logarithmic number system," *Computer and Information Science, 2008. ICIS 08. Seventh IEEE/ACIS International Conference on*, pp. 27–32, May 2008.

[99] S. C. Huang and L. G. Chen, "A LOG-EXP still image compression chip design," *Consumer Electronics, 1999. ICCE. International Conference on*, pp. 156–157, June 1999.

[100] S. C. Huang and L. G. Chen, "A LOG-EXP still image compression chip design," *Consumer Electronics, IEEE Transactions on*, vol. 45, no. 3, pp. 812–819, Aug 1999.

[101] S. C. Huang, L. G. Chen, and T. H. Chen, "The chip design of a 32-b logarithmic number system," *Circuits and Systems, 1994. ISCAS '94., 1994 IEEE International Symposium on*, vol. 4, pp. 167–170, May 1994.

[102] S. C. Huang, "Logarithmic number system processor for DSP applications," Master's thesis, National Taiwan University, 1993.

[103] M. Schulte and J. Stine, "Symmetric bipartite tables for accurate function approximation," *Computer Arithmetic, 1997. Proceedings., 13th IEEE Symposium on*, pp. 175–183, Jul 1997.

[104] M. Schulte and J. Stine, "Accurate function approximations by symmetric table lookup and addition," *Application-Specific Systems, Architectures and Processors, 1997. Proceedings., IEEE International Conference on*, pp. 144–153, Jul 1997.

[105] M. Schulte and J. Stine, "Approximating elementary functions with symmetric bipartite tables," *Computers, IEEE Transactions on*, vol. 48, no. 8, pp. 842–847, Aug 1999.

[106] G. Moore, "Cramming more components onto integrated circuits," *Electronics Magazine*, vol. 38, Apr 1965.

[107] D. Chen and S. B. Ko, "A dynamic non-uniform segmentation method for first-order polynomial function evaluation," *Microprocessors and Microsystems*, vol. 36, no. 4, pp. 324 – 332, 2012.

[108] S. F. Hsiao, H. J. Ko, and C. S. Wen, "Two-level hardware function evaluation based on correction of normalized piecewise difference functions," *Circuits and Systems II: Express Briefs, IEEE Transactions on*, vol. 59, no. 5, pp. 292–296, May 2012.

[109] J. Toler, "Apparatus using a logarithm based processor and an audio amplifier," U.S. Patent 5,948,052. Sep. 7 1999.

[110] F. S. Lai, "A 10 ns hybrid number system data execution unit for digital signal processing systems," *Solid-State Circuits, IEEE Journal of*, vol. 26, no. 4, pp. 590–599, Apr 1991.

[111] L. Pickett, "Method and apparatus for exponential/logarithmic computation," U.S. Patent 5,197,024. Mar. 23 1993.

[112] F. de Dinechin and A. Tisserand, "Multipartite table methods," *Computers, IEEE Transactions on*, vol. 54, no. 3, pp. 319–330, March 2005.

[113] H. Kim, B. G. Nam, J. H. Sohn, J. H. Woo, and H. J. Yoo, “A 231-mhz, 2.18-mw 32-bit logarithmic arithmetic unit for fixed-point 3-D graphics system,” *Solid-State Circuits, IEEE Journal of*, vol. 41, no. 11, pp. 2373–2381, Nov 2006.

[114] B. G. Nam, H. Kim, and H. J. Yoo, “Power and area-efficient unified computation of vector and elementary functions for handheld 3D graphics systems,” *Computers, IEEE Transactions on*, vol. 57, no. 4, pp. 490–504, April 2008.

[115] T. B. Juang, S. H. Chen, and H. J. Cheng, “A lower error and rom-free logarithmic converter for digital signal processing applications,” *Circuits and Systems II: Express Briefs, IEEE Transactions on*, vol. 56, no. 12, pp. 931–935, Dec 2009.

[116] S. Paul, N. Jayakumar, and S. Khatri, “A fast hardware approach for approximate, efficient logarithm and antilogarithm computations,” *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 17, no. 2, pp. 269–277, Feb 2009.

[117] D. De Caro, N. Petra, and A. Strollo, “Efficient logarithmic converters for digital signal processing applications,” *Circuits and Systems II: Express Briefs, IEEE Transactions on*, vol. 58, no. 10, pp. 667–671, Oct 2011.

[118] H. Fu, O. Mencer, and W. Luk, “Optimizing logarithmic arithmetic on FPGAs,” *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*, pp. 163–172, April 2007.

[119] M. Haselman, M. Beauchamp, A. Wood, S. Hauck, K. Underwood, and K. Hemmert, “A comparison of floating point and logarithmic number systems for FPGAs,” *Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE Symposium on*, pp. 181–190, April 2005.

[120] T. Sasao, S. Nagayama, and J. Butler, “Numerical function generators using LUT cascades,” *Computers, IEEE Transactions on*, vol. 56, no. 6, pp. 826–838, June 2007.

[121] D. U. Lee, R. C. C. Cheung, W. Luk, and J. Villasenor, "Hardware implementation trade-offs of polynomial approximations and interpolations," *Computers, IEEE Transactions on*, vol. 57, no. 5, pp. 686–701, May 2008.

[122] D. U. Lee, A. Gaffar, O. Mencer, and W. Luk, "Optimizing hardware function evaluation," *Computers, IEEE Transactions on*, vol. 54, no. 12, pp. 1520–1531, Dec 2005.

[123] E. C. Ifeachor and B. W. Jervis, *Digital Signal Processing: A Practical Approach*, 2nd ed. Pearson Education, 2002.

[124] M. Rae, "Applications of correlation techniques," *Rheologica Acta*, vol. 8, no. 2, pp. 157–161, 1969.

[125] X. Qian, "In-line measurement and characterisation of pneumatically conveyed pulverised coal and biomass using non-intrusive electrostatic sensor arrays," Ph.D. dissertation, University of Kent, 2012.

[126] X. Qian, Y. Yan, J. Shao, LijuanWang, H. Zhou, and C. Wang, "Quantitative characterization of pulverized coal and biomasscoal blends in pneumatic conveying pipelines using electrostatic sensor arrays and data fusion techniques," *Measurement science and technology*, vol. 23, p. 13, 2012.

[127] J. Shao, J. Krabicka, and Y. Yan, "Velocity measurement of pneumatically conveyed particles using intrusive electrostatic sensors," *Instrumentation and Measurement, IEEE Transactions on*, vol. 59, no. 5, pp. 1477–1484, May 2010.

[128] Y. Yan, Z. Xie, J. Krabicka, and J. Shao, "Non-contact strip speed measurement using electrostatic sensors," *Instrumentation and Measurement Technology Conference (I2MTC), 2010 IEEE*, pp. 1535–1538, May 2010.

[129] Y. Yan, S. J. Rodrigues, and Z. Xie, "Non-contact strip speed measurement using electrostatic sensing and correlation signal-processing techniques," *Measurement Science and Technology*, vol. 22, p. 9, 2011.

[130] C. Tan and F. Dong, "Cross correlation velocity of oil-water two-phase flow by a dual-plane electrical resistance tomography system," *Instrumentation and*

*Measurement Technology Conference (I2MTC), 2010 IEEE*, pp. 766–770, May 2010.

[131] Y. Lin, C. Huang, D. Irwin, L. He, Y. Shang, and G. Yu, "Three-dimensional flow contrast imaging of deep tissue using noncontact diffuse correlation tomography," *Applied Physics Letters*, vol. 104, no. 12, pp. 121 103–121 103–4, Mar 2014.

[132] N. Ayob, S. Yaacob, Z. Zakaria, M. Fazalul Rahiman, and R. Rahim, "Simulation on using cross-correlation technique for two-phase liquid/gas flow measurement for ultrasonic transmission tomography," *Signal Processing and Its Applications (CSPA), 2010 6th International Colloquium on*, pp. 1–5, May 2010.

[133] X. Wu, X. Wu, and Z. Zhu, "Tilting alignment for electron tomography based on local cross-correlation and adaptive searching," *IT in Medicine and Education, 2008. ITME 2008. IEEE International Symposium on*, pp. 684–688, Dec 2008.

[134] J. Munoz-Gomez, J. Bartrina-Rapesta, M. Marcellin, and J. Serra-Sagrista, "Correlation modeling for compression of computed tomography images," *Biomedical and Health Informatics, IEEE Journal of*, vol. 17, no. 5, pp. 928–935, Sept 2013.

[135] M. Dabbicco, V. Spagnolo, M. Troccoli, C. Marinelli, and G. Scamarcio, "Correlation between laser pattern and local carrier distribution in vcsels determined by microprobe electroluminescence," *Lasers and Electro-Optics Europe, 2000. Conference Digest. 2000 Conference on*, p. 1, Sept 2000.

[136] H. Miyajima, N. Shigei, and Y. Hamakawa, "Higher order differential correlation associative memory of sequential patterns," *Neural Networks, 2004. Proceedings. 2004 IEEE International Joint Conference on*, vol. 2, pp. 891–896 vol.2, July 2004.

[137] D. Walvoord, K. Baum, M. Helguera, A. Krol, and R. Easton, "Localization of fiducial skin markers in mr images using correlation pattern recognition for

pet/mri nonrigid breast image registration," *Applied Imagery Pattern Recognition Workshop, 2008. AIPR '08. 37th IEEE*, pp. 1–4, Oct 2008.

[138] H. El-Bakry and Q. Zhao, "Fast pattern detection using neural networks and cross correlation in the frequency domain," *Neural Networks, 2005. IJCNN '05. Proceedings. 2005 IEEE International Joint Conference on*, vol. 3, pp. 1900–1905, July 2005.

[139] J. Carlson and R.-K. Ing, "Ultrasonic particle velocimetry in multiphase flows," *Ultrasonics Symposium, 2002. Proceedings. 2002 IEEE*, vol. 1, pp. 761–764, Oct 2002.

[140] W. Zhang, C. Wang, and Y. Wang, "Parameter selection in cross-correlation-based velocimetry using circular electrostatic sensors," *Instrumentation and Measurement, IEEE Transactions on*, vol. 59, no. 5, pp. 1268–1275, May 2010.

[141] C. Huang, J. Shen, and X. Cai, "Optical correlation velocimetry: An aid to particle sizing by acoustic emission measurement," *Systems and Informatics (ICSAI), 2012 International Conference on*, pp. 2633–2635, May 2012.

[142] D. Hill, S. Young, P. Parker, and E. Pike, "Photon correlation velocimetry of blood flow in the retina," *Quantum Electronics, IEEE Journal of*, vol. 13, no. 9, pp. 897–898, September 1977.

[143] S. Yatsushiro, K. Kuroda, A. Hirayama, H. Atsumi, and M. Matsumae, "Correlation time mapping based on magnetic resonance velocimetry: Preliminary results on cerebrospinal fluid flow," *Biomedical Engineering International Conference (BMEiCON), 2013 6th*, pp. 1–4, Oct 2013.

[144] S. Yaofeng, T. Y. Meng, J. Pang, and S. Fei, "Digital image correlation and its applications in electronics packaging," *Electronic Packaging Technology Conference, 2005. EPTC 2005. Proceedings of 7th*, vol. 1, p. 6, Dec 2005.

[145] L. Siegel, H. Siegel, and A. Feather, "Parallel processing approaches to image correlation," *Computers, IEEE Transactions on*, vol. C-31, no. 3, pp. 208–218, March 1982.

[146] X. Xianming, H. Wenxiang, and W. Hongru, “Digital image correlation method (dicm) application in speckle phase-shift of shear speckle defect detection,” *Intelligent Signal Processing and Communication Systems (ISPACS), 2010 International Symposium on*, pp. 1–4, Dec 2010.

[147] J. Fan, A. Kot, H. Cao, and F. Sattar, “Modeling the exif-image correlation for image manipulation detection,” *Image Processing (ICIP), 2011 18th IEEE International Conference on*, pp. 1945–1948, Sept 2011.

[148] G. Guo and C. Dyer, “Patch-based image correlation with rapid filtering,” *Computer Vision and Pattern Recognition, 2007. CVPR ’07. IEEE Conference on*, pp. 1–6, June 2007.

[149] M. Cavadini, M. Wosnitza, and G. Troster, “Multiprocessor system for high-resolution image correlation in real time,” *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 9, no. 3, pp. 439–449, June 2001.

[150] Y. Wang, J. Liu, and D. Li, “Study on correlation of micro stereovision with stereo light microscope,” *Mechatronics and Automation, 2007. ICMA 2007. International Conference on*, pp. 948–952, Aug 2007.

[151] S. Lefebvre, S. Ambellouis, and F. Cabestaing, “Obstacles detection on a road by dense stereovision with 1D correlation windows and fuzzy filtering,” *Intelligent Transportation Systems Conference, 2006. ITSC ’06. IEEE*, pp. 739–744, Sept 2006.

[152] J. Marie-Julie, P. Adam, and D. Juvin, “Real time stereovision using correlation on a parallel simd computer, sympati 2,” *Algorithms and Architectures for Parallel Processing, 1995. ICAPP 95. IEEE First ICA/sup 3/PP., IEEE First International Conference on*, vol. 1, pp. 422–426, Apr 1995.

[153] L. Boissier, B. Hotz, C. Proy, O. Faugeras, and P. Fua, “Autonomous planetary rover (vap): on-board perception system concept and stereovision by correlation approach,” *Robotics and Automation, 1992. Proceedings., 1992 IEEE International Conference on*, pp. 181–186, May 1992.

[154] M. Ouali, H. Lange, and C. Laurgeau, “An energy minimization approach to dense stereovision,” *Image Processing, 1996. Proceedings., International Conference on*, vol. 1, pp. 841–845, Sep 1996.

[155] D. Lyon, “The discrete fourier transform, part 6: Cross-correlation,” *Journal of object technology*, vol. 9, pp. 17–22, 2010.

[156] I. R. Barratt, Y. Yan, and B. Byrne, “A parallel-beam radiometric instrumentation system for the mass flow measurement of pneumatically conveyed solids,” *Measurement Science and Technology*, vol. 12, pp. 1515–1528, 2001.

[157] L. Wang and Y. Yan, “Mathematical modelling and experimental validation of electrostatic sensors for rotational speed measurement,” *Measurement Science and Technology*, vol. 25, no. 11, p. 115101, 2014.

[158] L. Wang, Y. Yan, Y. Hu, and X. Qian, “Rotational speed measurement through electrostatic sensing and correlation signal processing,” *Instrumentation and Measurement, IEEE Transactions on*, vol. 63, no. 5, pp. 1190–1199, May 2014.

[159] J. Urena, “Correlation detector based on a FPGA for ultrasonic sensors,” *Microprocessors and Microsystems*, vol. 23, pp. 25–33, 1999.

[160] W. Xie, Y. Zhou, and L. Li, “Application of phase only correlation in velocity measurement based on FPGA,” *Image and Signal Processing (CISP), 2011 4th International Congress on*, vol. 3, pp. 1301–1304, Oct 2011.

[161] A. HajiRassouliha, T. Gamage, M. Parker, M. Nash, A. Taberner, and P. Nielsen, “FPGA implementation of 2D cross-correlation for real-time 3D tracking of deformable surfaces,” *Image and Vision Computing New Zealand (IVCNZ), 2013 28th International Conference of*, pp. 352–357, Nov 2013.

[162] J. Ding, X. Du, X. Wang, and J. Liu, “Improved real-time correlation-based FPGA stereo vision system,” *Mechatronics and Automation (ICMA), 2010 International Conference on*, pp. 104–108, Aug 2010.

[163] S. H. Jin, J. U. Cho, D. R. Lee, J. H. Park, H. S. Kim, C. H. Lee, J. S. Choi, and J. W. Jeon, "An FPGA based voice signal preprocessor for the real-time cross-correlation," *Control, Automation and Systems, 2007. ICCAS '07. International Conference on*, pp. 793–797, Oct 2007.

[164] G. Danese, M. Giachero, F. Leporati, G. Matrone, and N. Nazzicari, "An FPGA based embedded system for fingerprint matching using phase only correlation algorithm," *Digital System Design, Architectures, Methods and Tools, 2009. DSD '09. 12th Euromicro Conference on*, pp. 672–679, Aug 2009.

[165] P. Lee, K. Adefila, and Y. Yan, "An FPGA correlator for continuous real-time measurement of particulate flow," *Instrumentation and Measurement Technology Conference (I2MTC), 2012 IEEE International*, pp. 2183–2186, May 2012.

[166] "AD7476/AD7477/AD7478: 1 MSPS, 12-/10-/8-Bit ADCs in 6-Lead SOT-23 Data Sheet (Rev F, 02/2009)," Analog Devices, One Technology Way, P.O. Box 9106, Norwood, MA 02062-9106, U.S.A.

[167] "DAC121S101/DAC121S101Q 12-Bit Micro Power, RRO Digital-to-Analog Converter ," Texas Instruments, Texas Instruments, Post Office Box 655303, Dallas, Texas 75265, U.S.A.

# Appendix A

# Table 4.2 Matlab Simulations

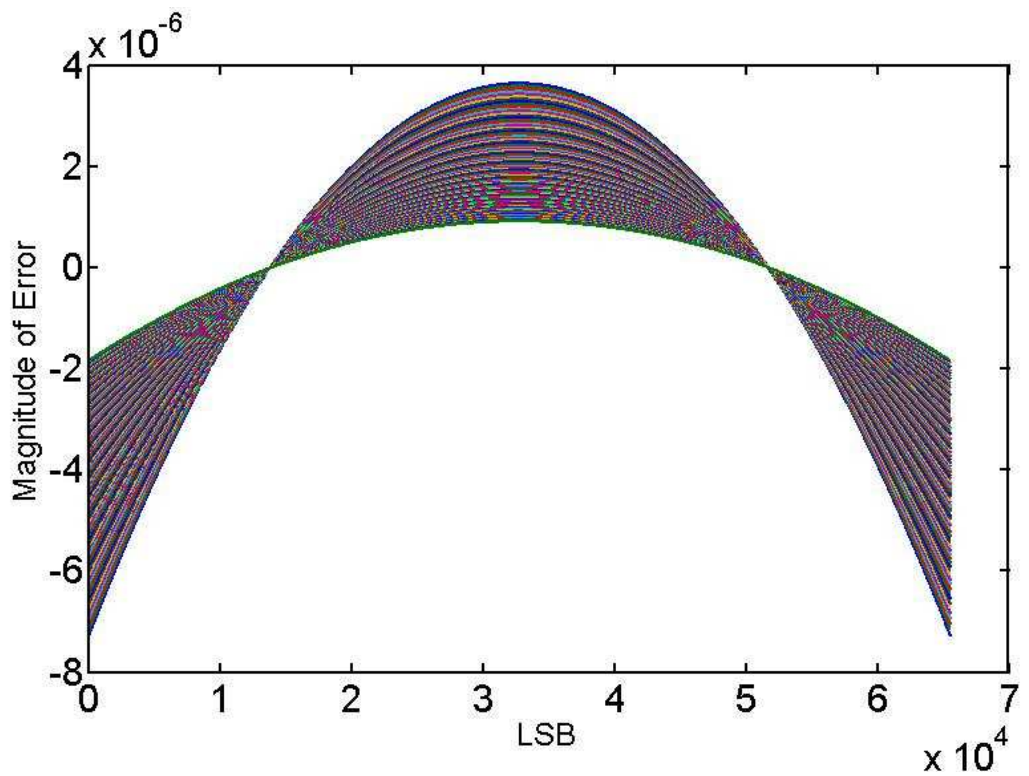


Figure A.1: Configuartion 7 : 16 :: 7 : 9 - 1st stage errors for each segment.

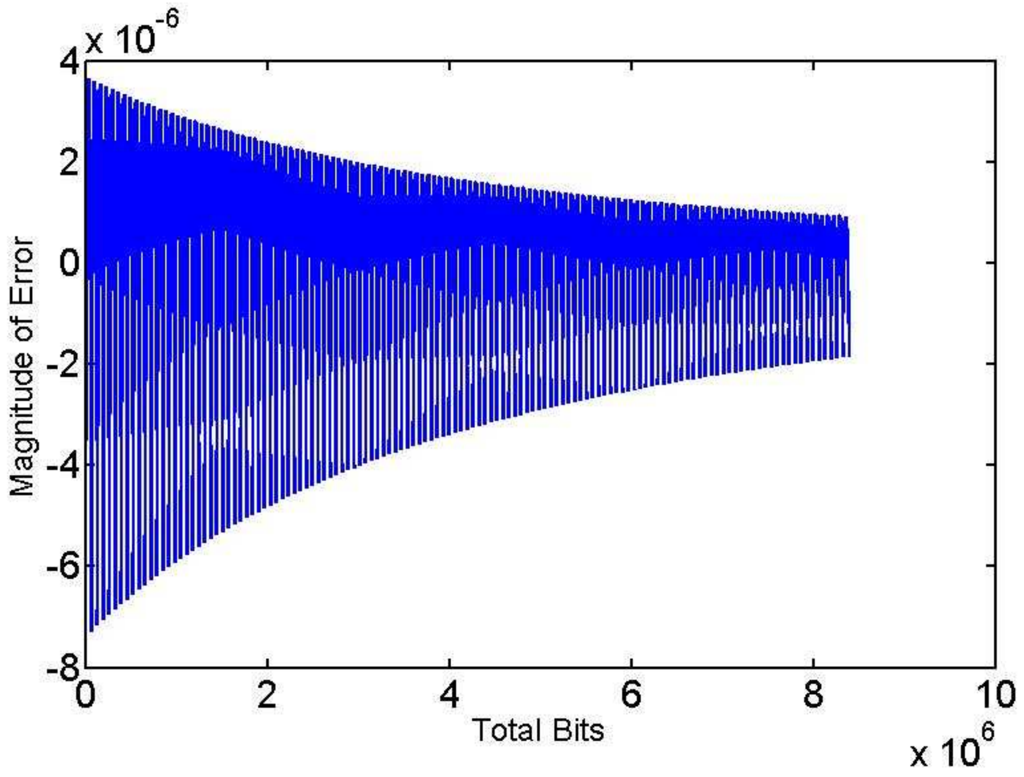


Figure A.2: Configuartion 7 : 16 :: 7 : 9 - 1st stage error approximation.

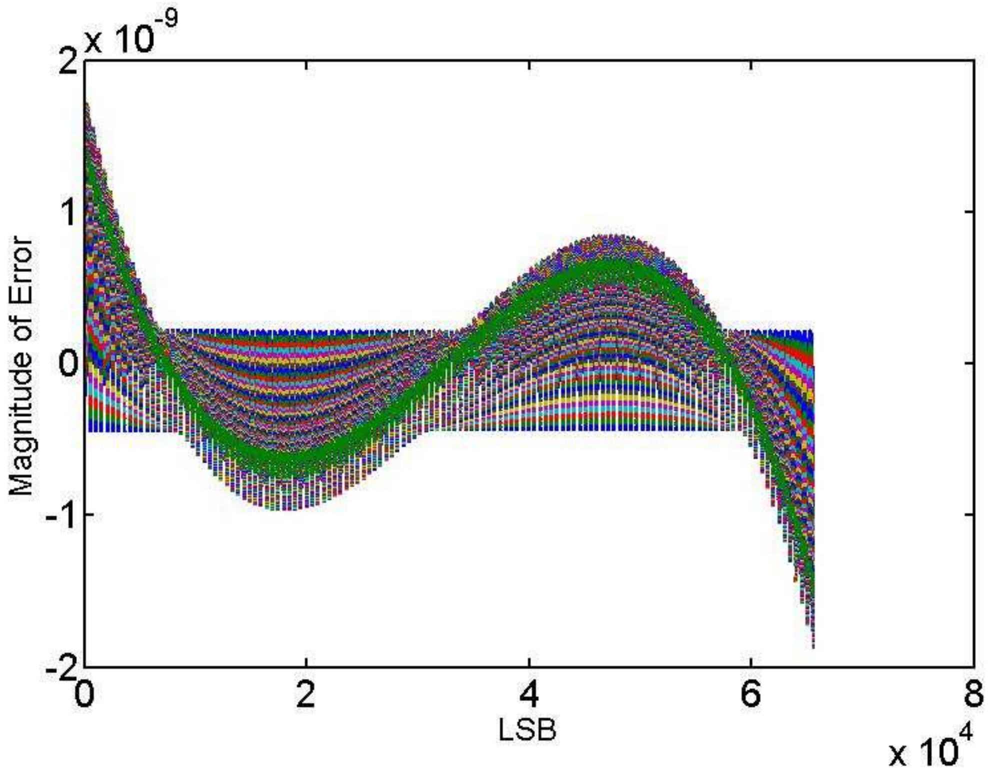


Figure A.3: Configuartion 7 : 16 :: 7 : 9 - 1st stage + 2nd stage errors for each segment.

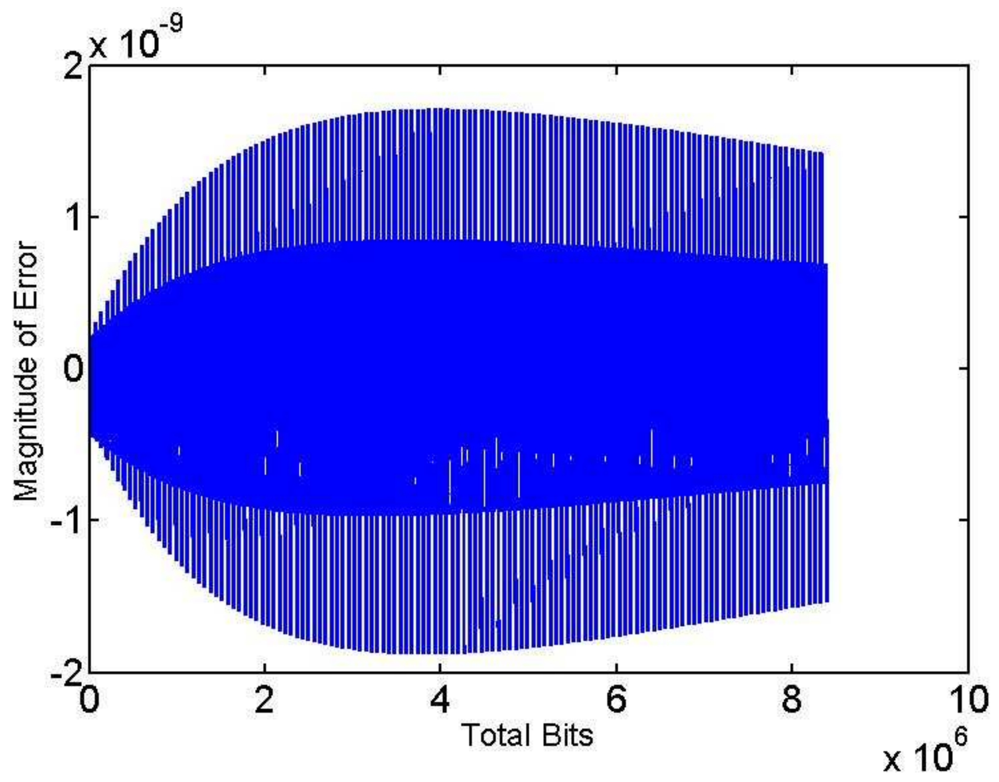


Figure A.4: Configuartion 7 : 16 :: 7 : 9 - 1st stage + 2nd stage error approximation.

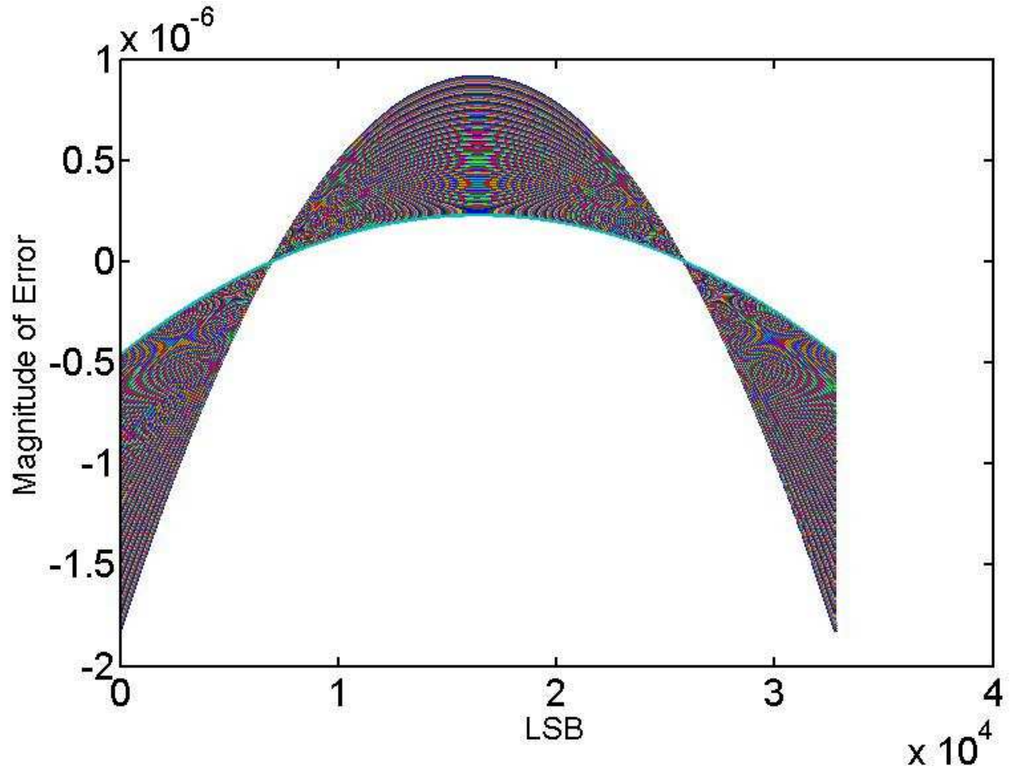


Figure A.5: Configuartion 8 : 15 :: 6 : 9 - 1st stage errors for each segment.

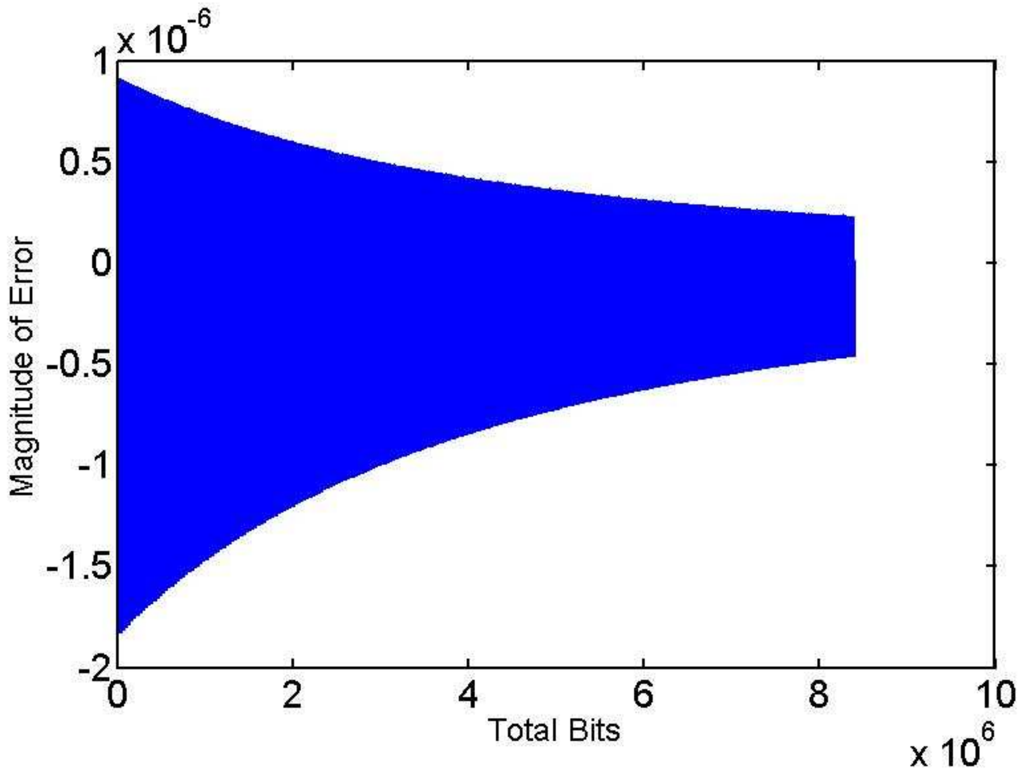


Figure A.6: Configuartion 8 : 15 :: 6 : 9 - 1st stage error approximation.

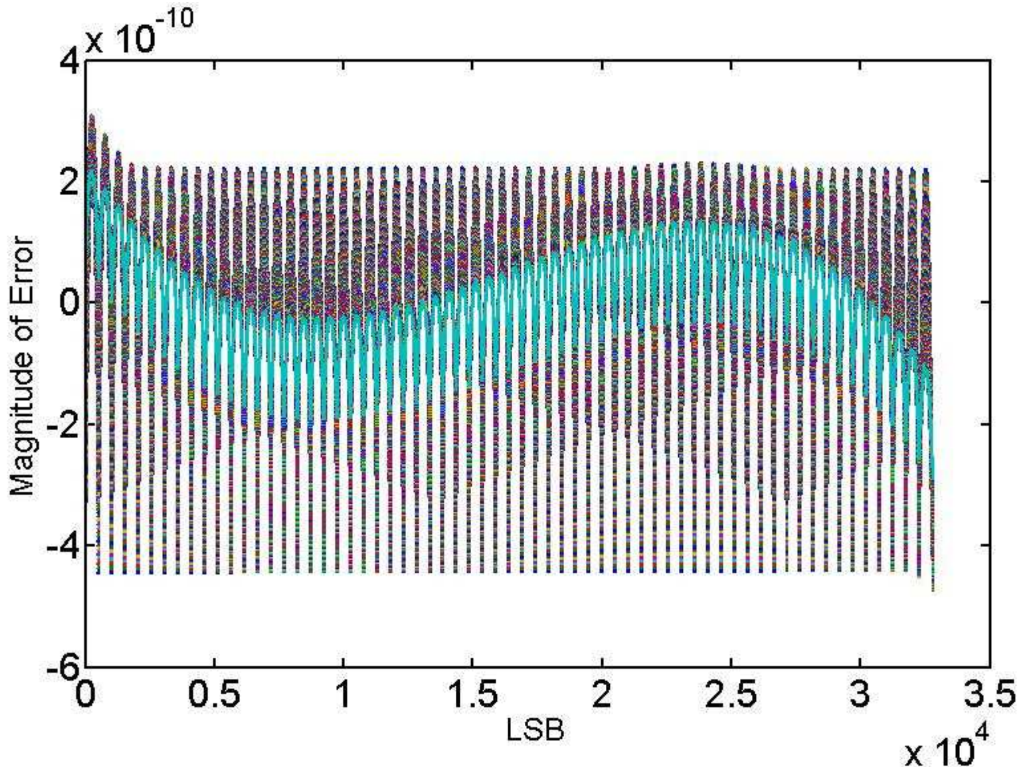


Figure A.7: Configuartion 8 : 15 :: 6 : 9 - 1st stage + 2nd stage errors for each segment.

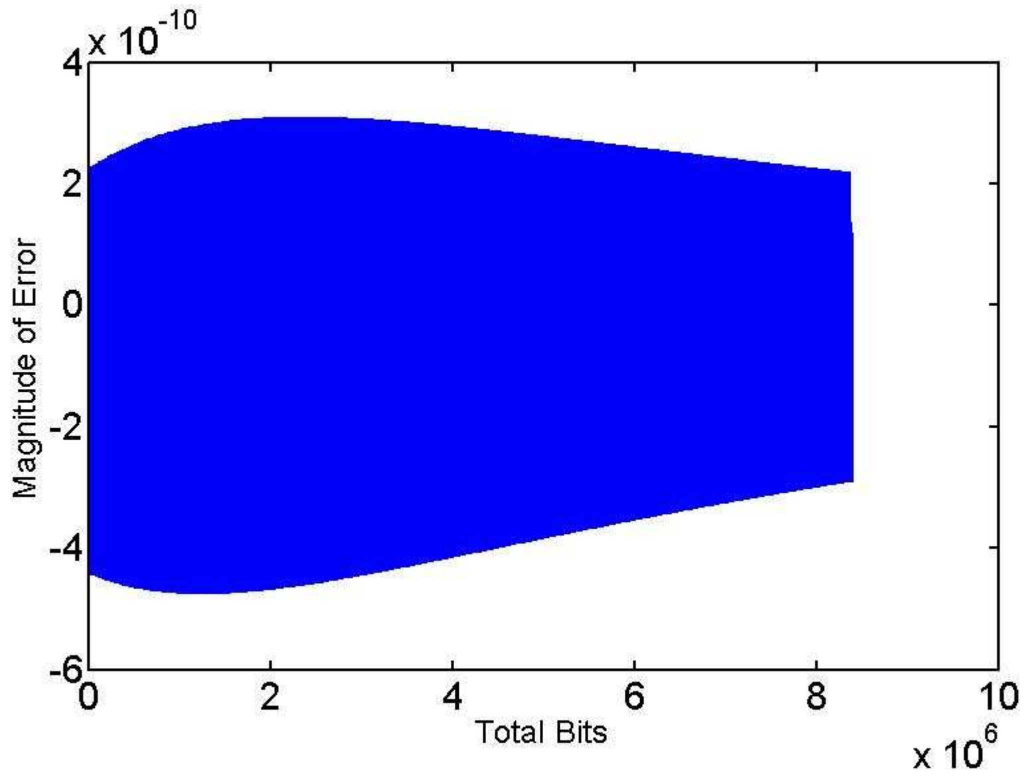


Figure A.8: Configuartion 8 : 15 :: 6 : 9 - 1st stage + 2nd stage error approximation.

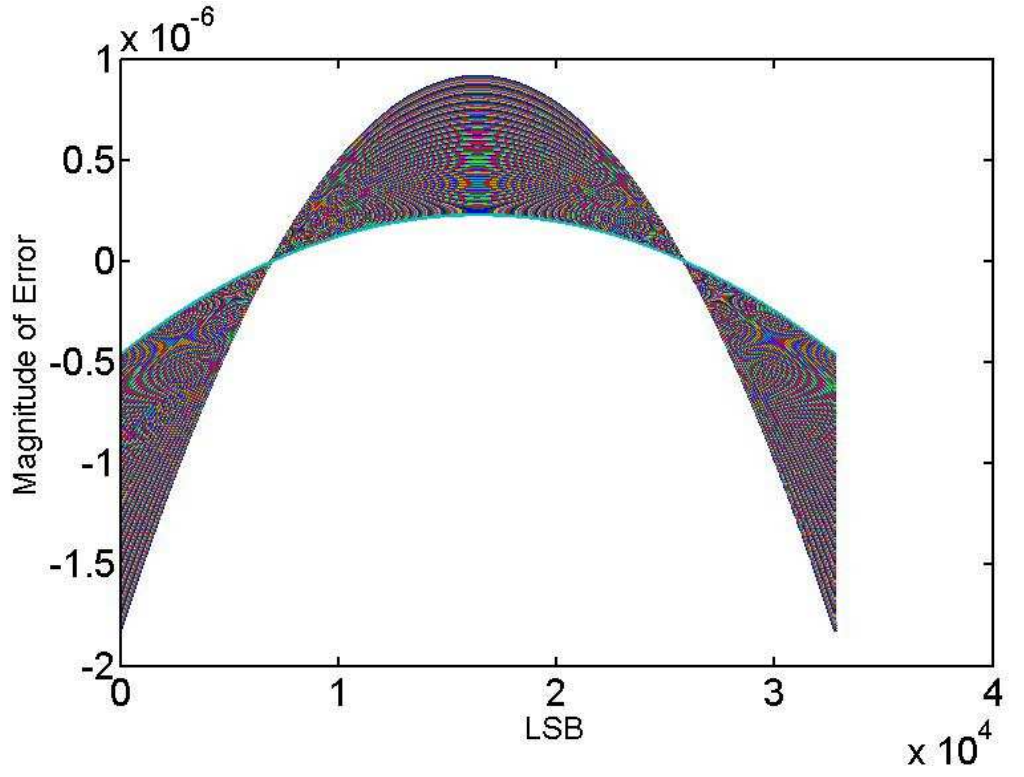


Figure A.9: Configuartion 8 : 15 :: 7 : 8 - 1st stage errors for each segment.

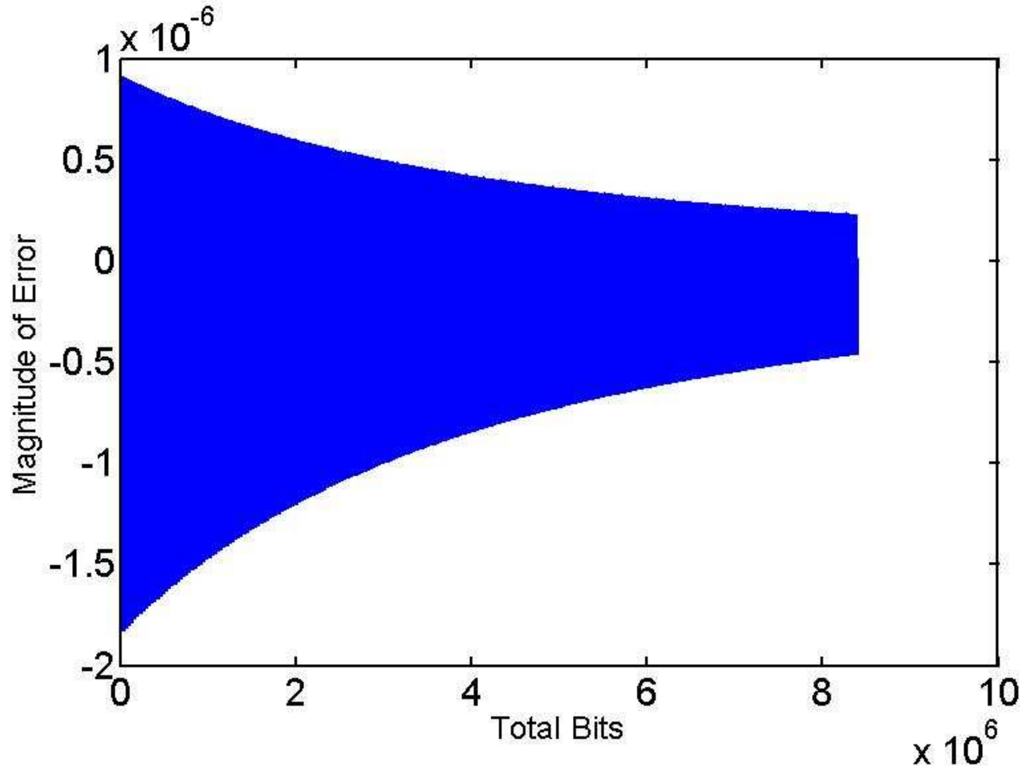


Figure A.10: Configuartion 8 : 15 :: 7 : 8 - 1st stage error approximation.

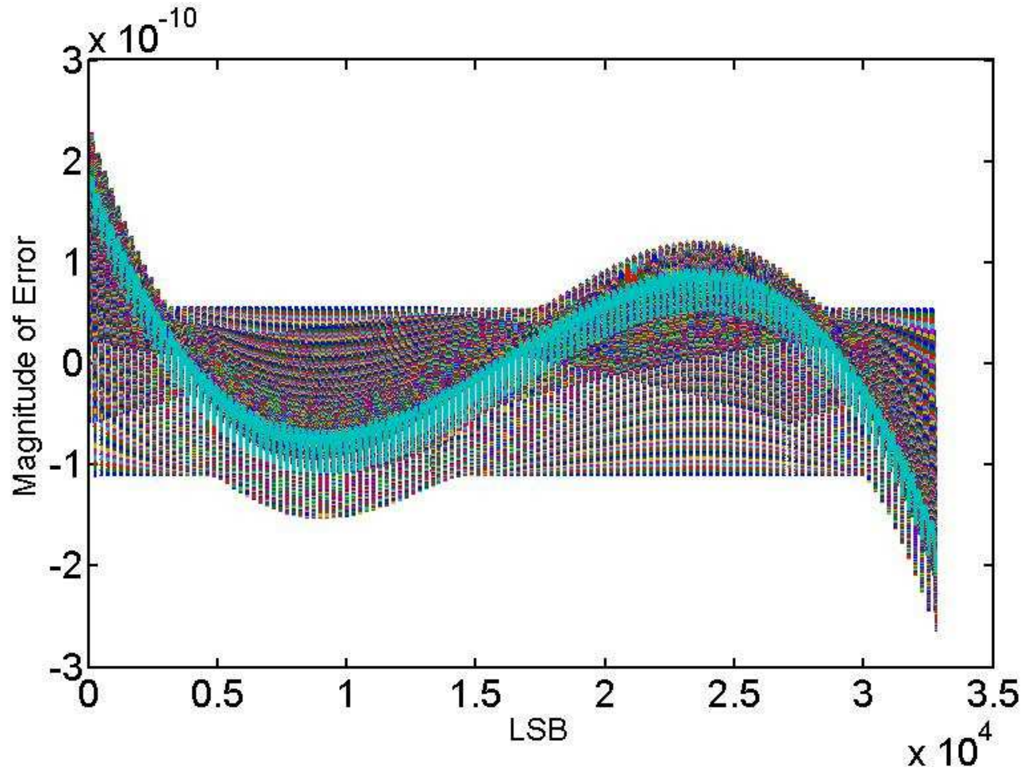


Figure A.11: Configuartion 8 : 15 :: 7 : 8 - 1st stage + 2nd stage errors for each segment.

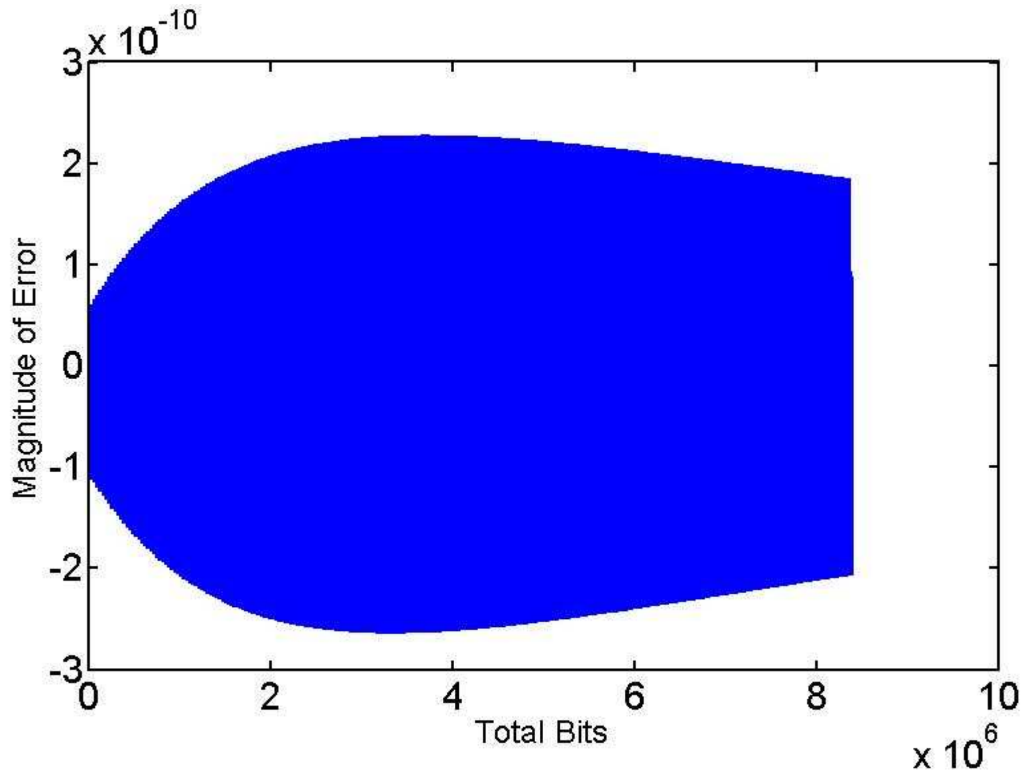


Figure A.12: Configuartion 8 : 15 :: 7 : 8 - 1st stage + 2nd stage error approximation.

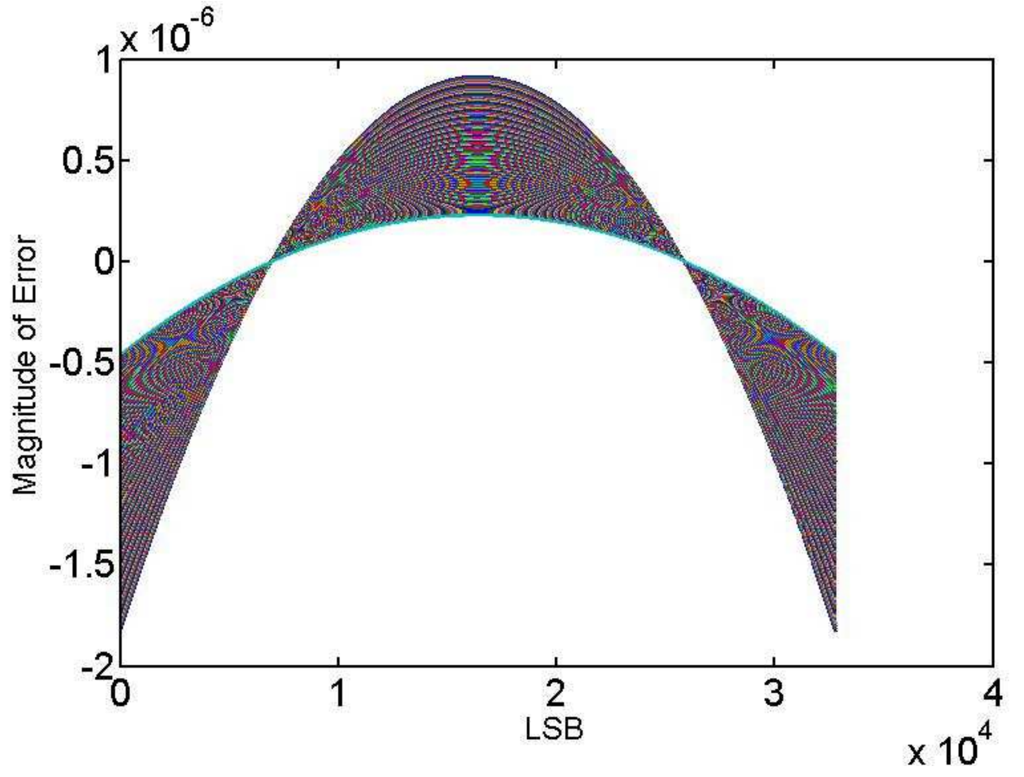


Figure A.13: Configuartion 8 : 15 :: 8 : 7 - 1st stage errors for each segment.

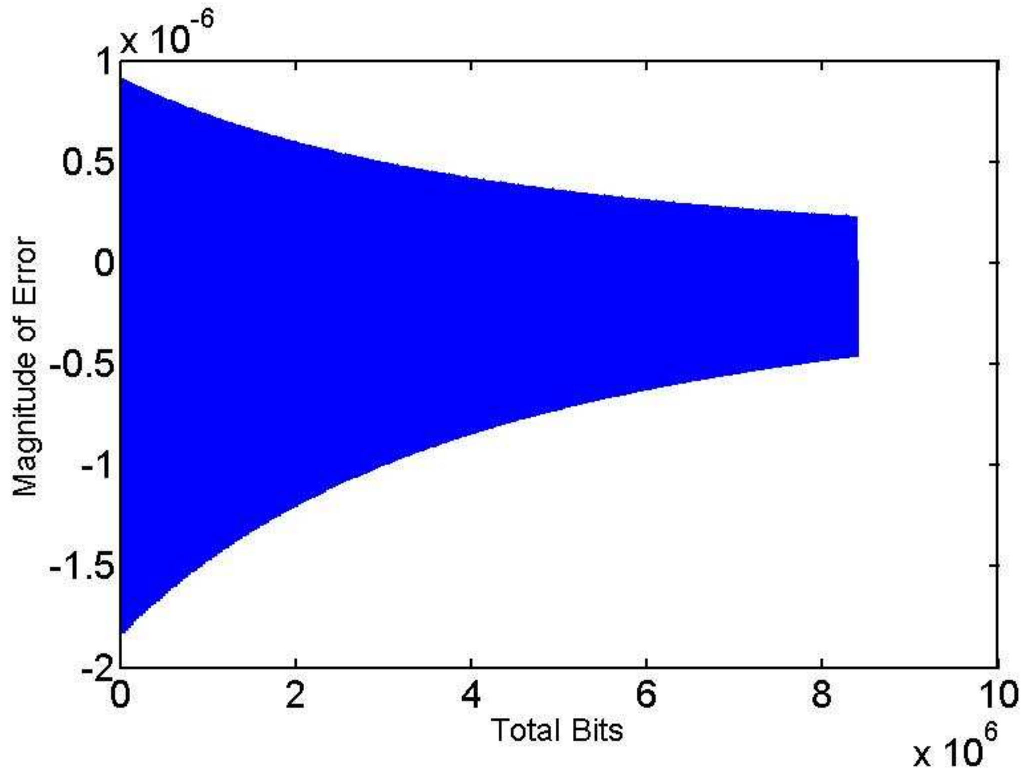


Figure A.14: Configuartion 8 : 15 :: 8 : 7 - 1st stage error approximation.

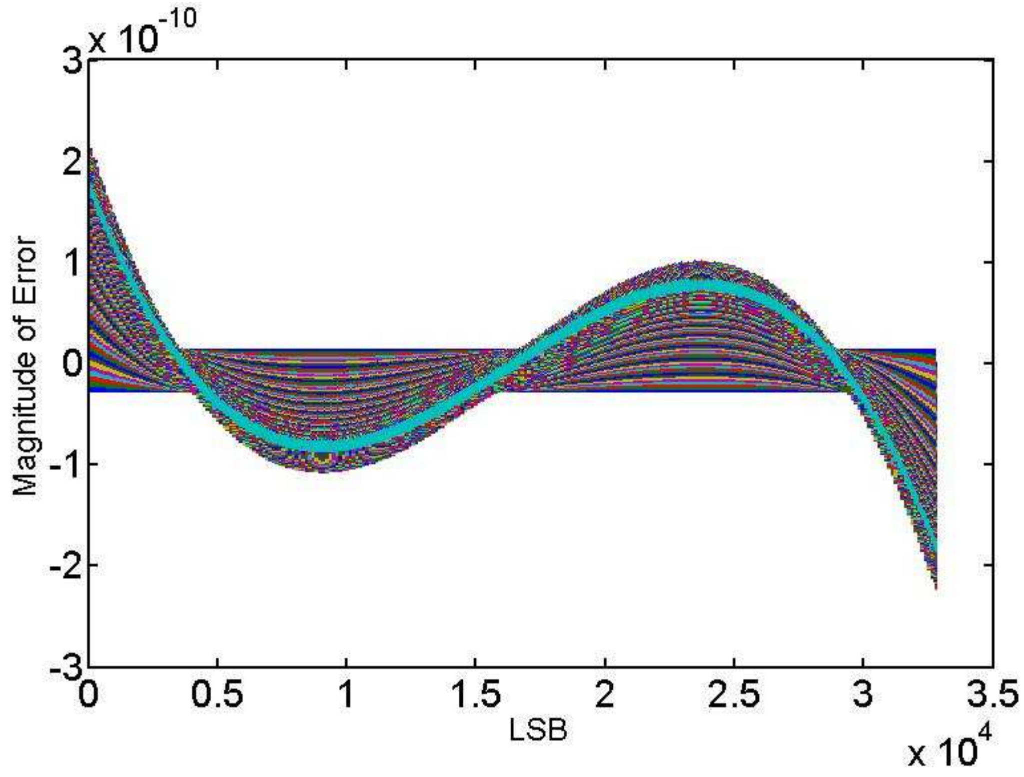


Figure A.15: Configuartion 8 : 15 :: 8 : 7 - 1st stage + 2nd stage errors for each segment.

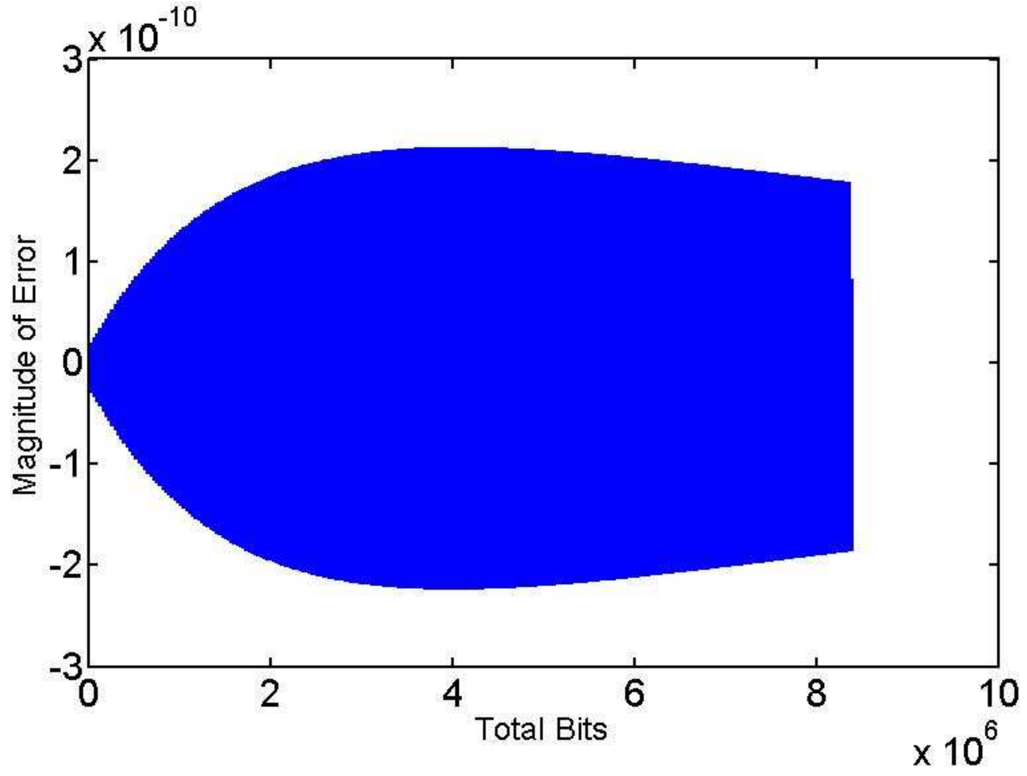


Figure A.16: Configuartion 8 : 15 :: 8 : 7 - 1st stage + 2nd stage error approximation.

# Appendix B

# Table 4.3 Matlab Simulations

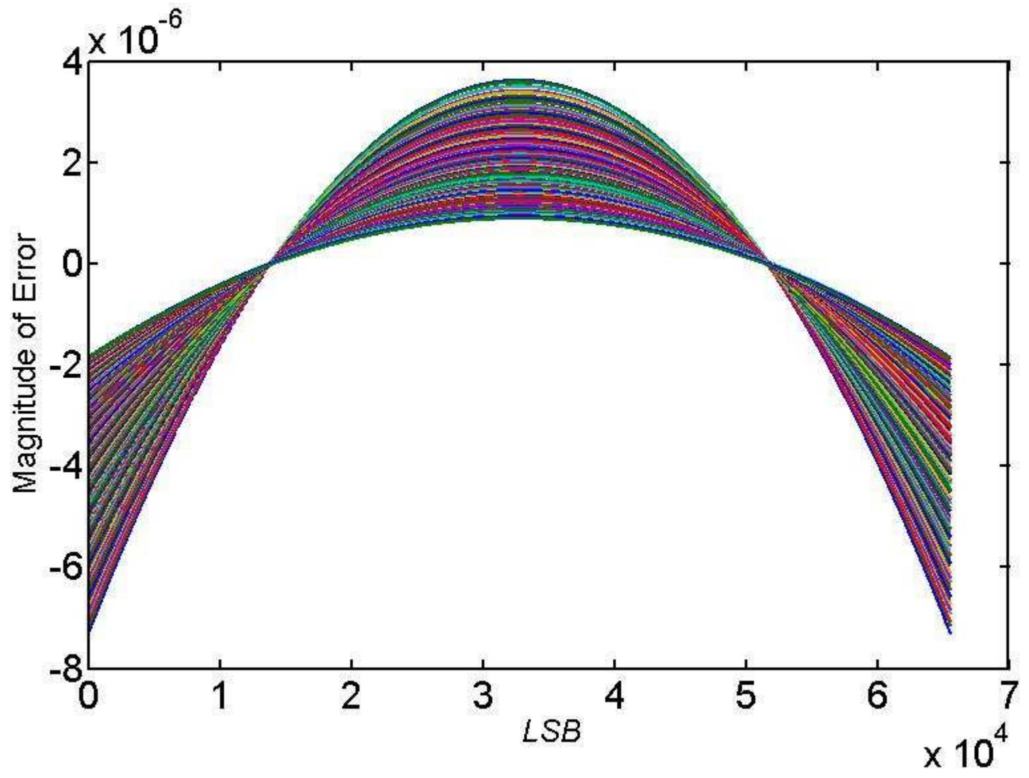


Figure B.1: Configuartion 7 : 16 :: 7 : 9 - 1st stage errors for each segment.

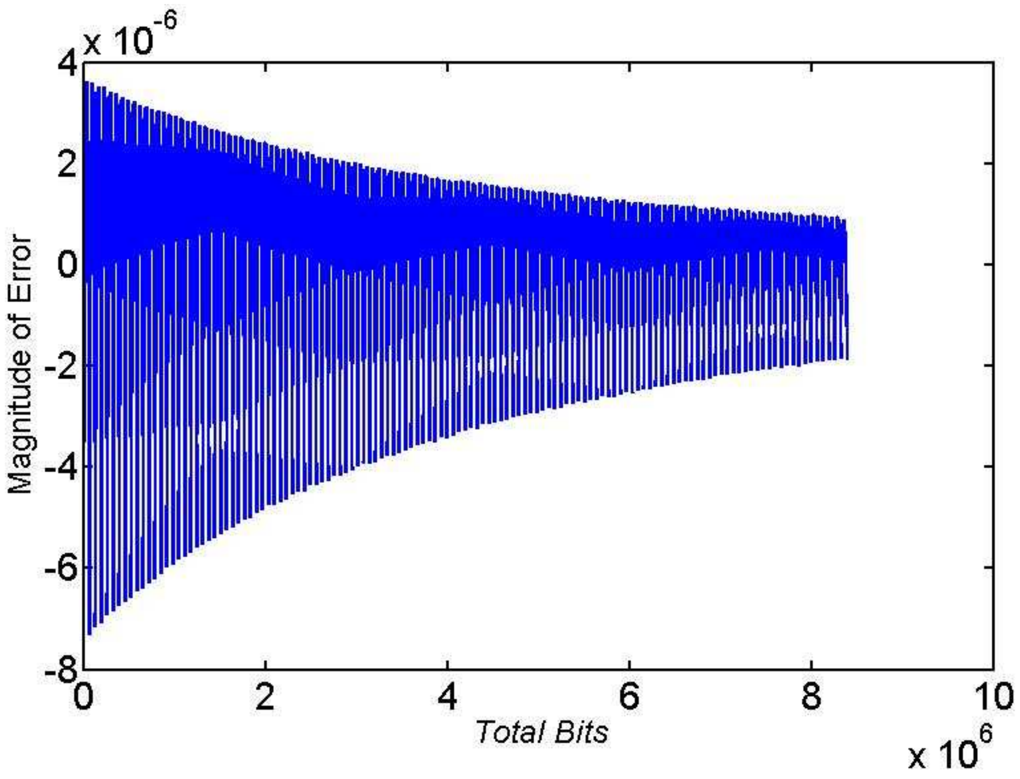


Figure B.2: Configuartion 7 : 16 :: 7 : 9 - 1st stage error approximation.

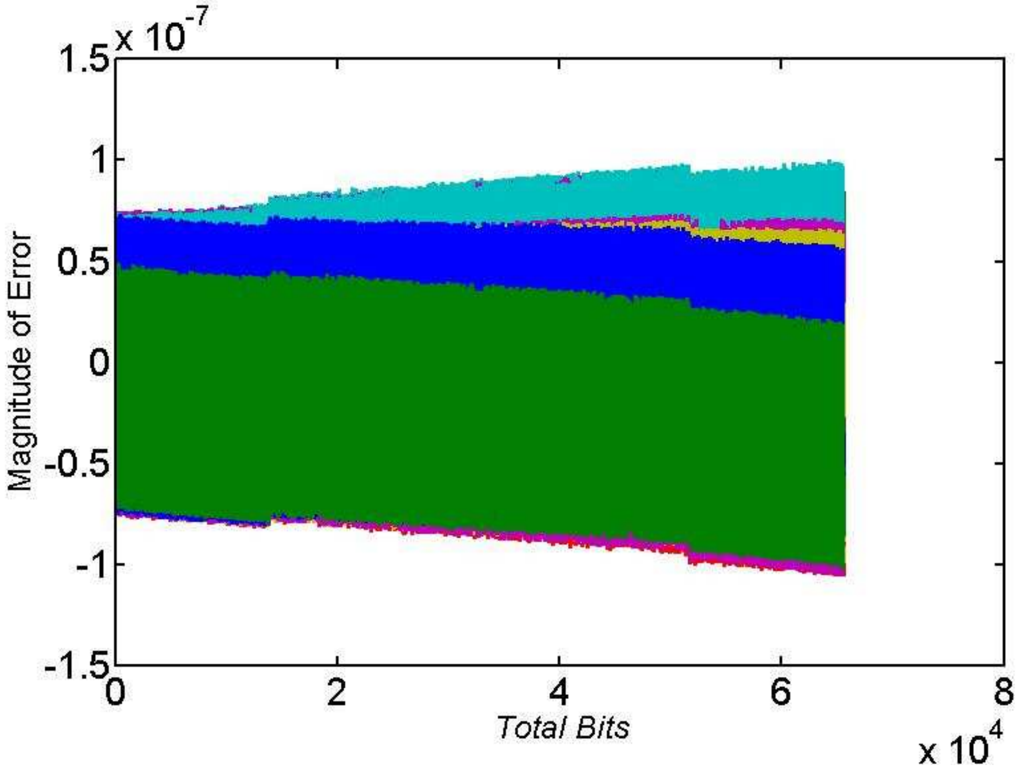


Figure B.3: Configuartion 7 : 16 :: 7 : 9 - Residual error in normalised error curve.

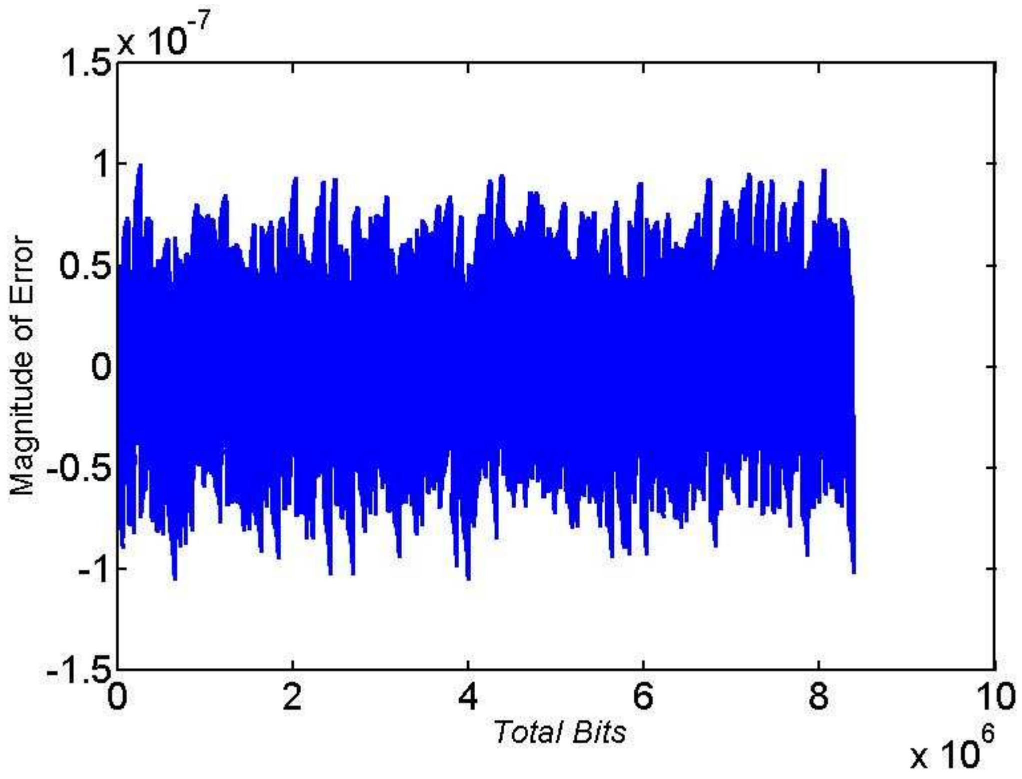


Figure B.4: Configuartion 7 : 16 :: 7 : 9 - 1st stage + 2nd stage error approximation.

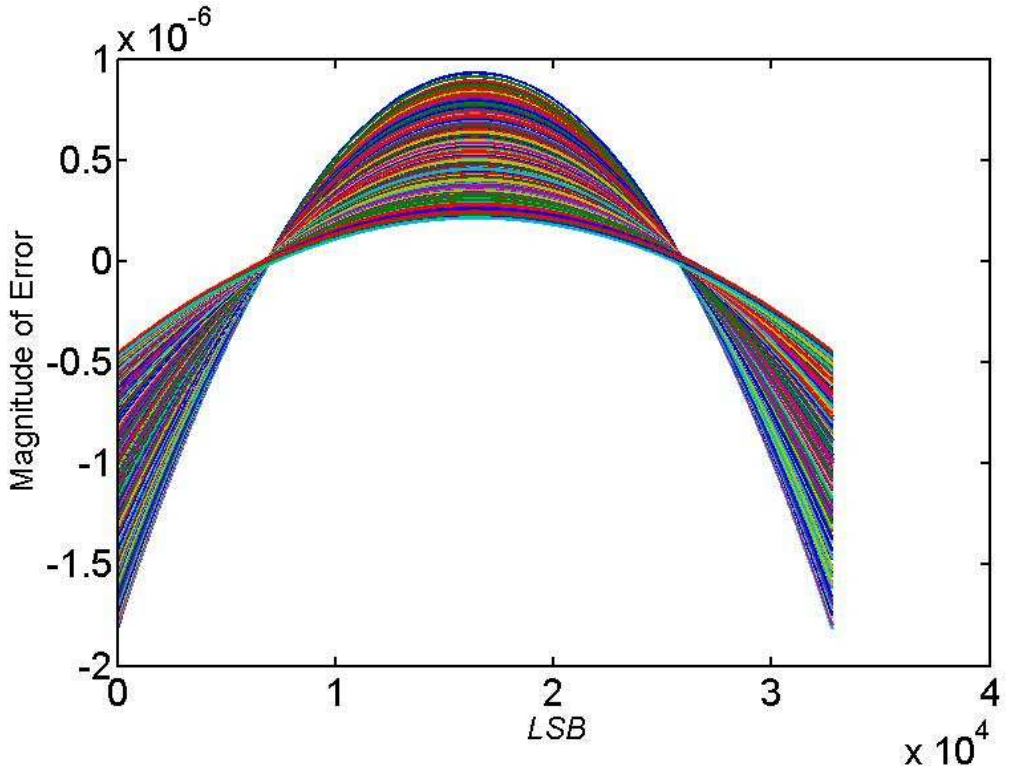


Figure B.5: Configuartion 8 : 15 :: 6 : 9 - 1st stage errors for each segment.

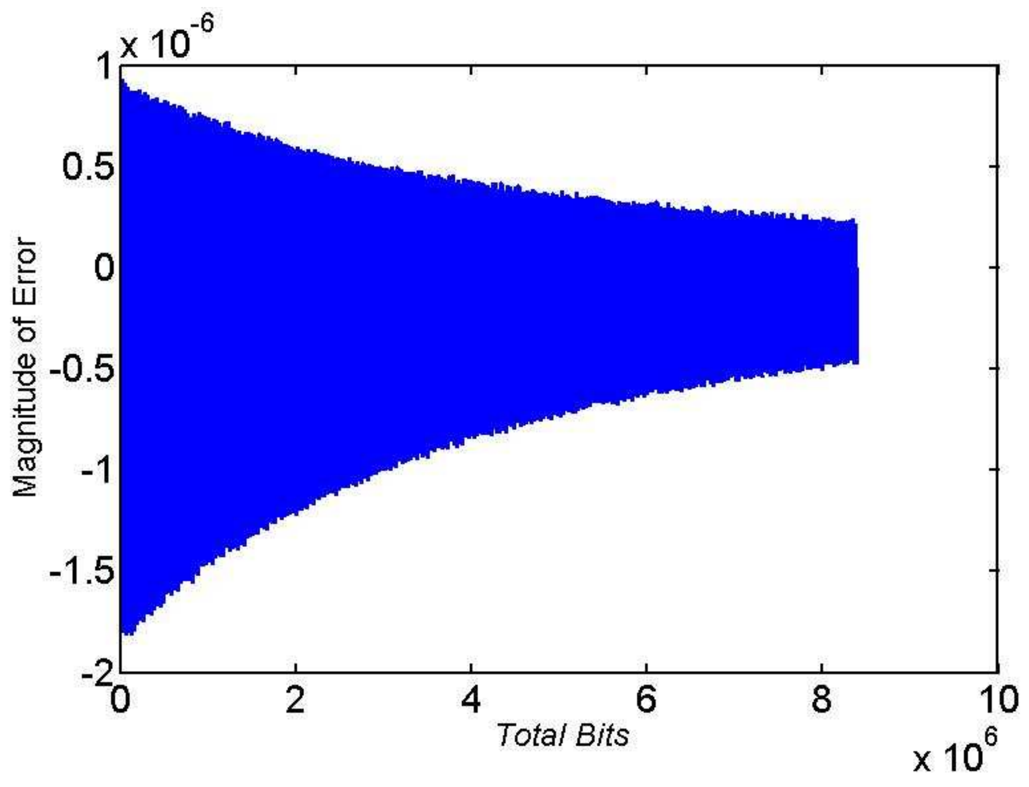


Figure B.6: Configuartion 8 : 15 :: 6 : 9 - 1st stage error approximation.

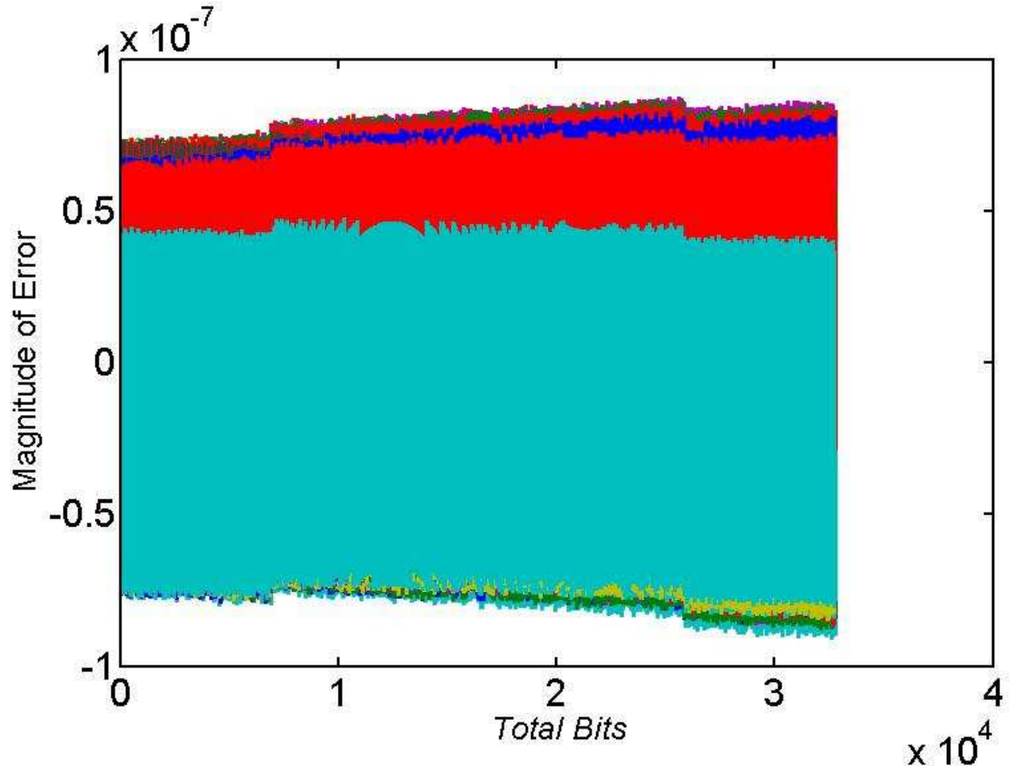


Figure B.7: Configuartion 8 : 15 :: 6 : 9 - Residual error in normalised error curve.

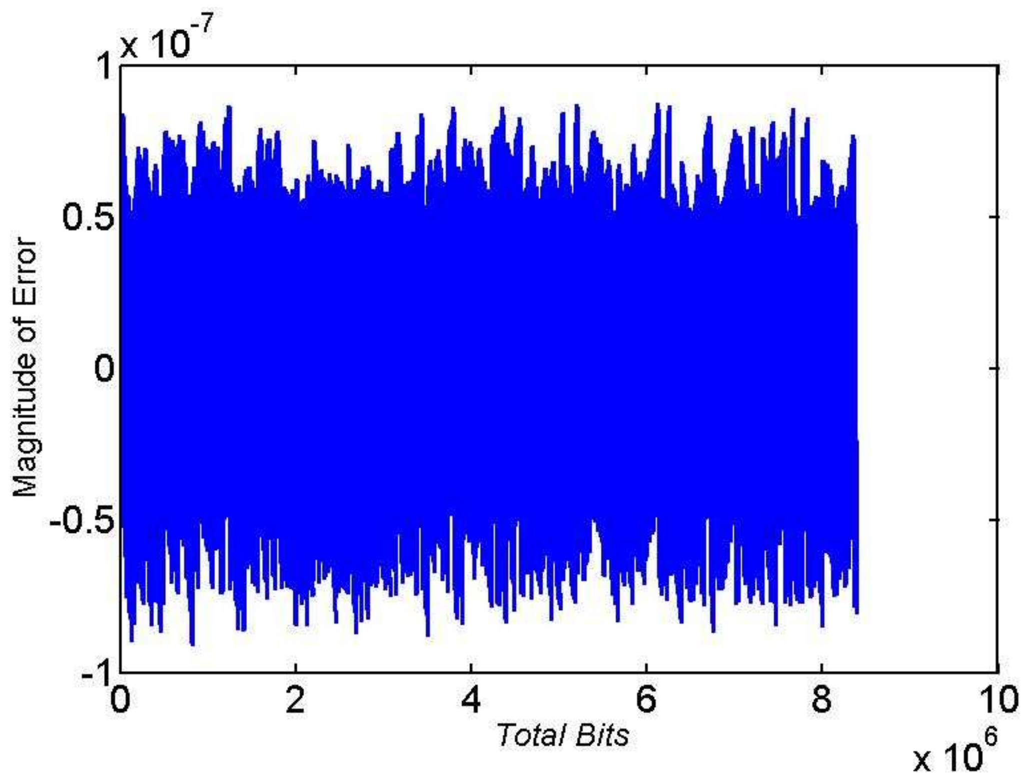


Figure B.8: Configuartion 8 : 15 :: 6 : 9 - 1st stage + 2nd stage error approximation.

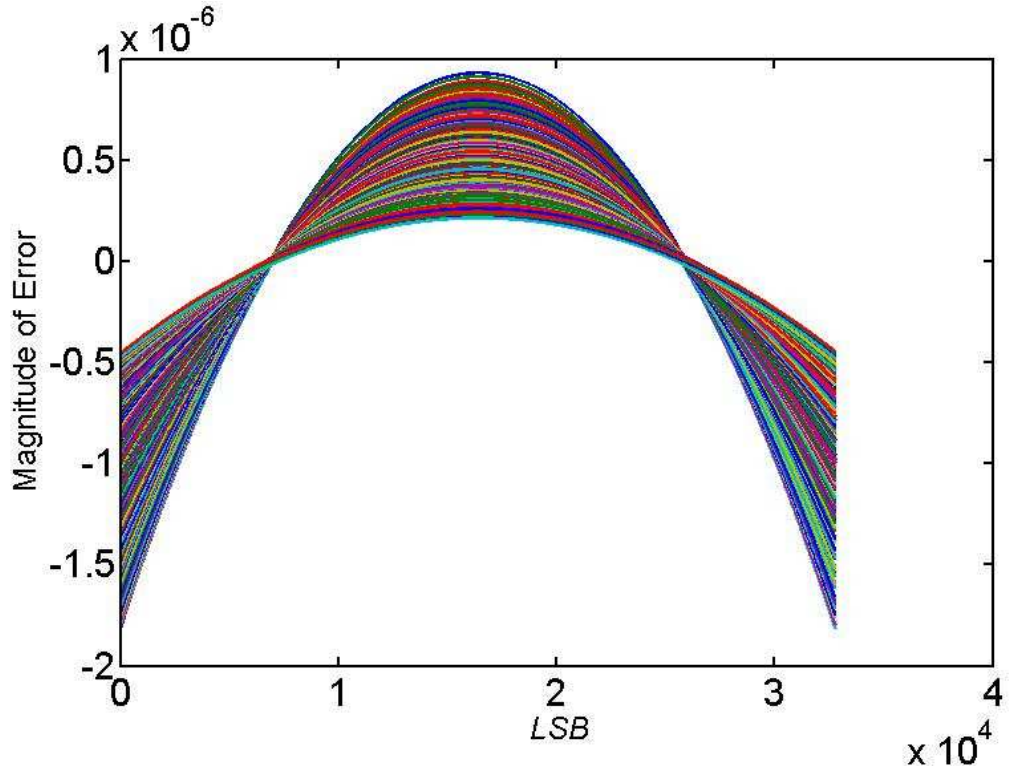


Figure B.9: Configuartion 8 : 15 :: 7 : 8 - 1st stage errors for each segment.

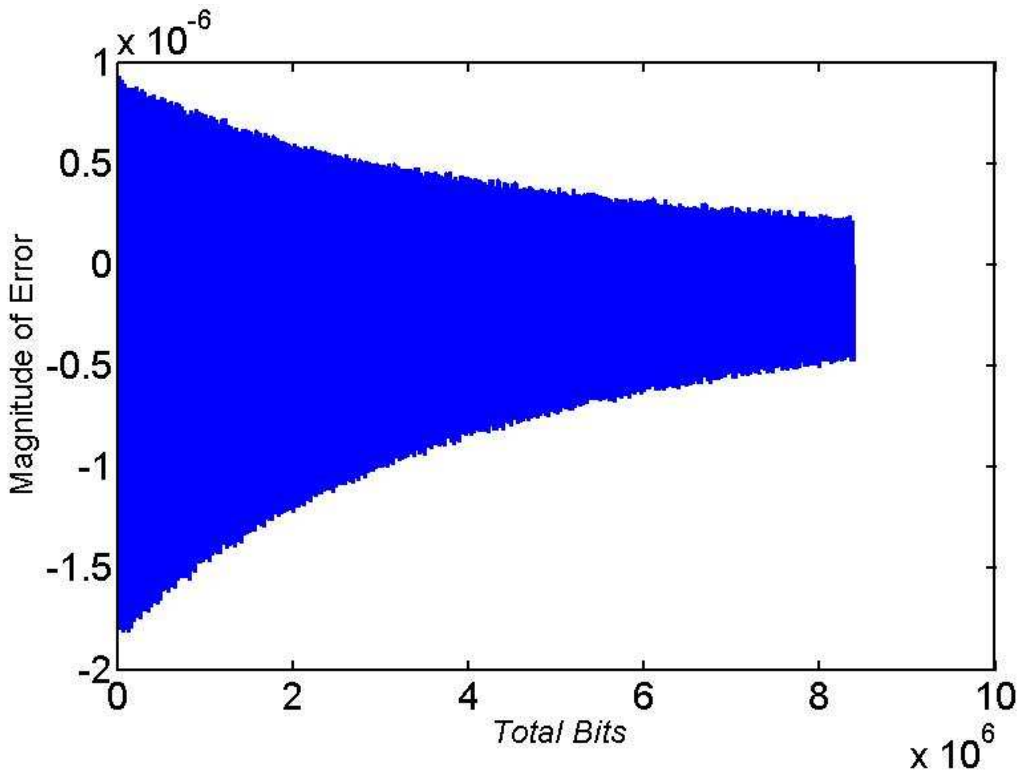


Figure B.10: Configuartion 8 : 15 :: 7 : 8 - 1st stage error approximation.

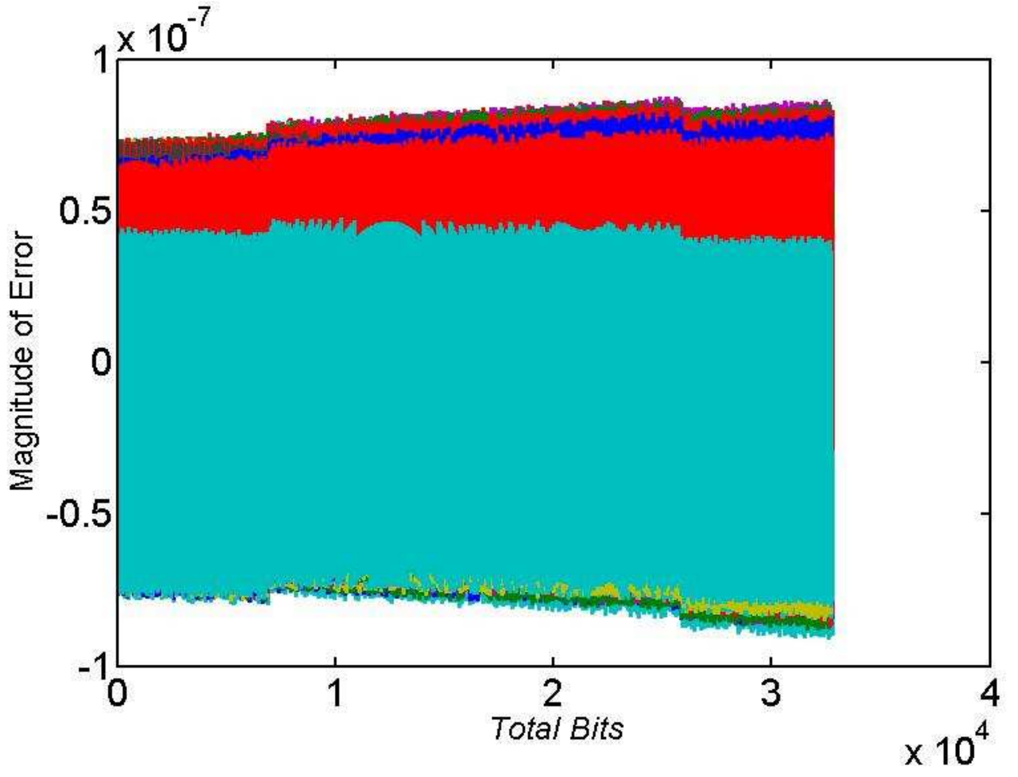


Figure B.11: Configuartion 8 : 15 :: 7 : 8 - Residual error in normalised error curve.

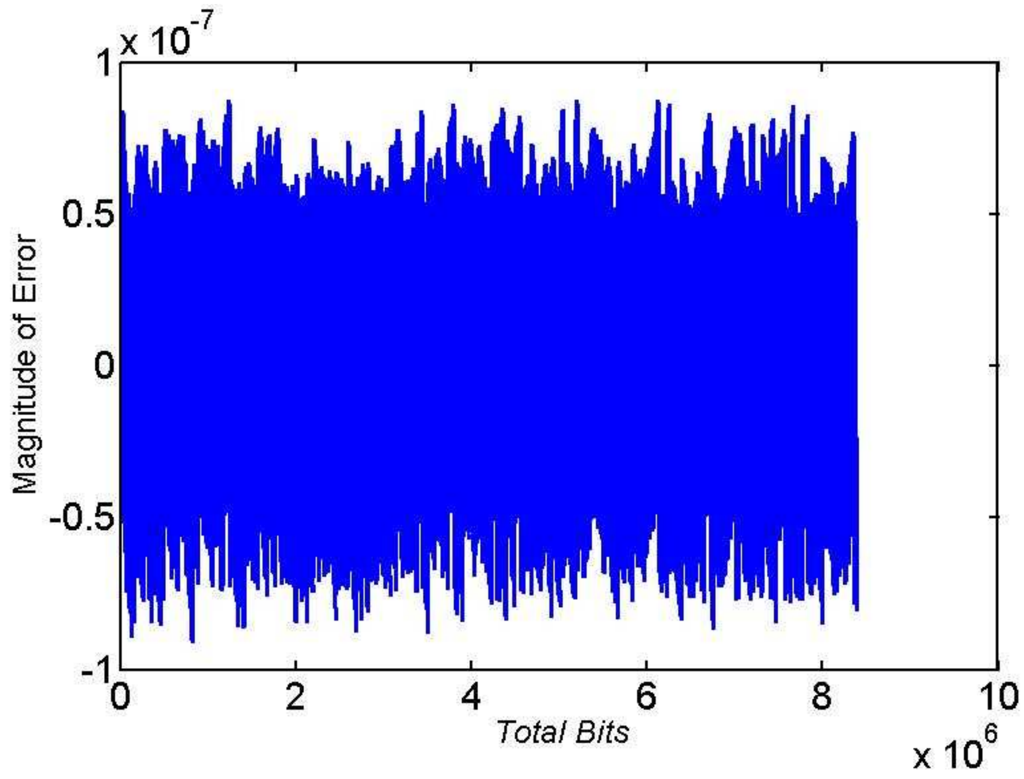


Figure B.12: Configuartion 8 : 15 :: 7 : 8 - 1st stage + 2nd stage error approximation.

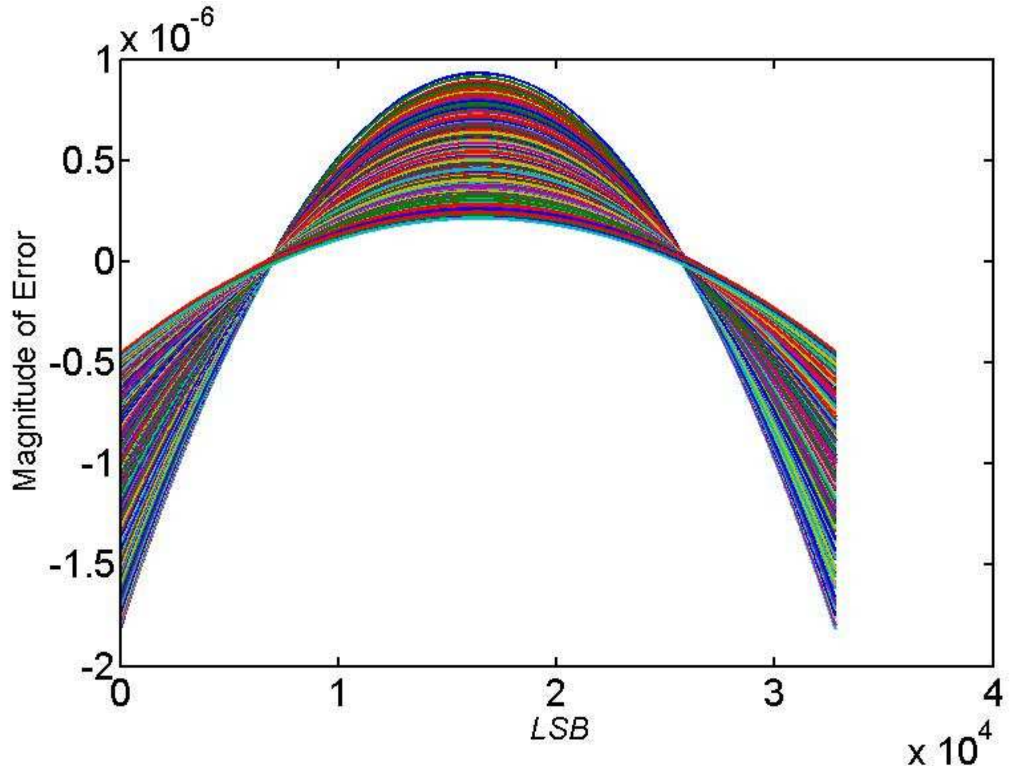


Figure B.13: Configuartion 8 : 15 :: 8 : 7 - 1st stage errors for each segment.

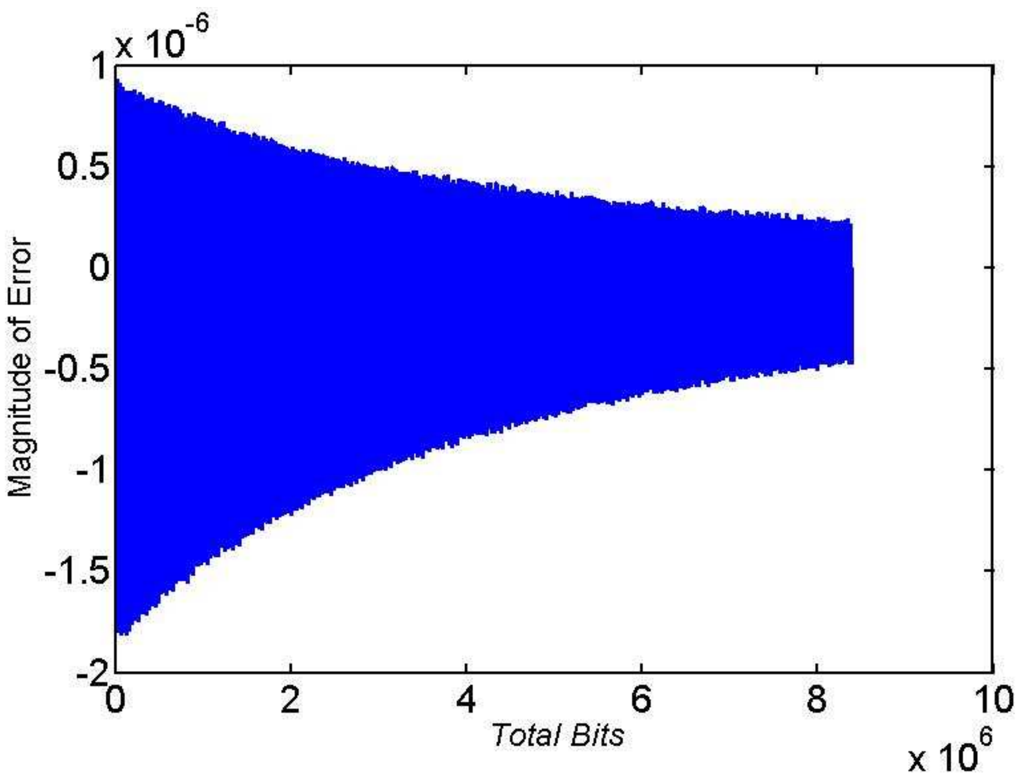


Figure B.14: Configuartion 8 : 15 :: 8 : 7 - 1st stage error approximation.

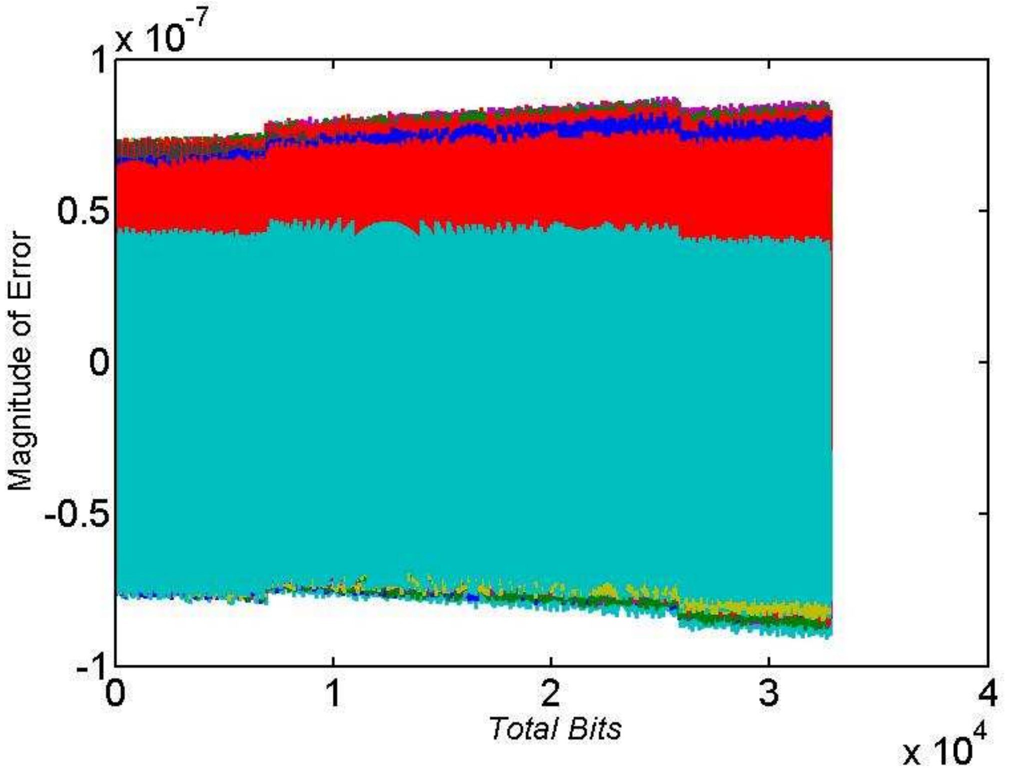


Figure B.15: Configuartion 8 : 15 :: 8 : 7 - Residual error in normalised error curve.

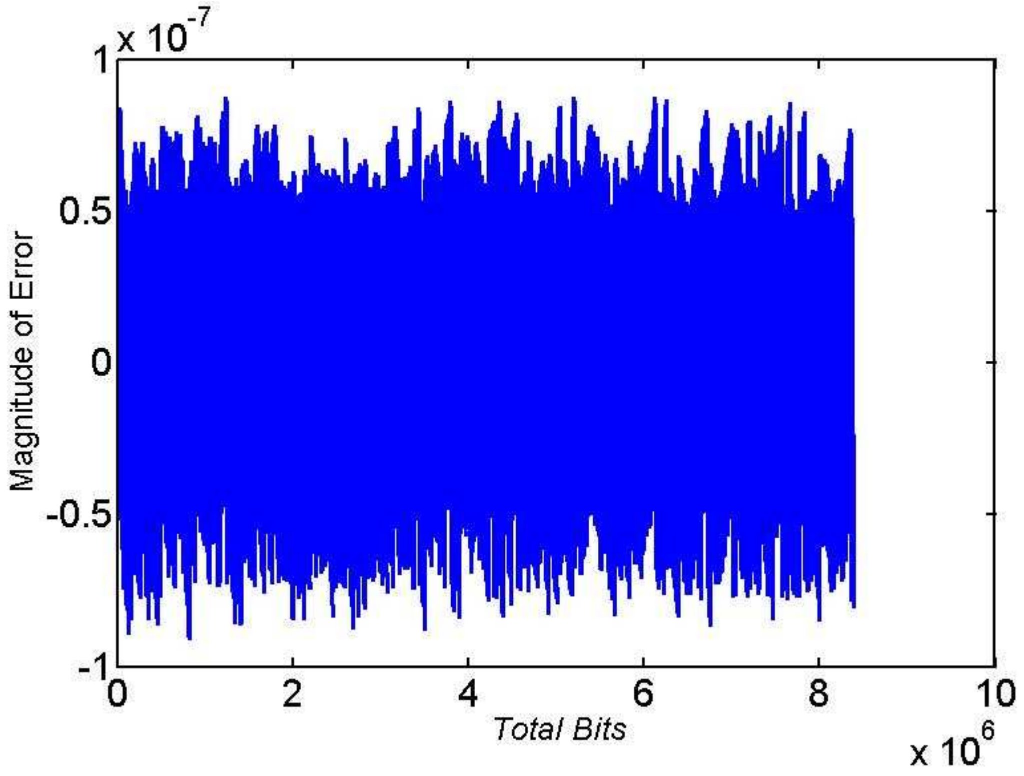


Figure B.16: Configuartion 8 : 15 :: 8 : 7 - 1st stage + 2nd stage error approximation.

# Appendix C

# Table 5.2 Matlab Simulations

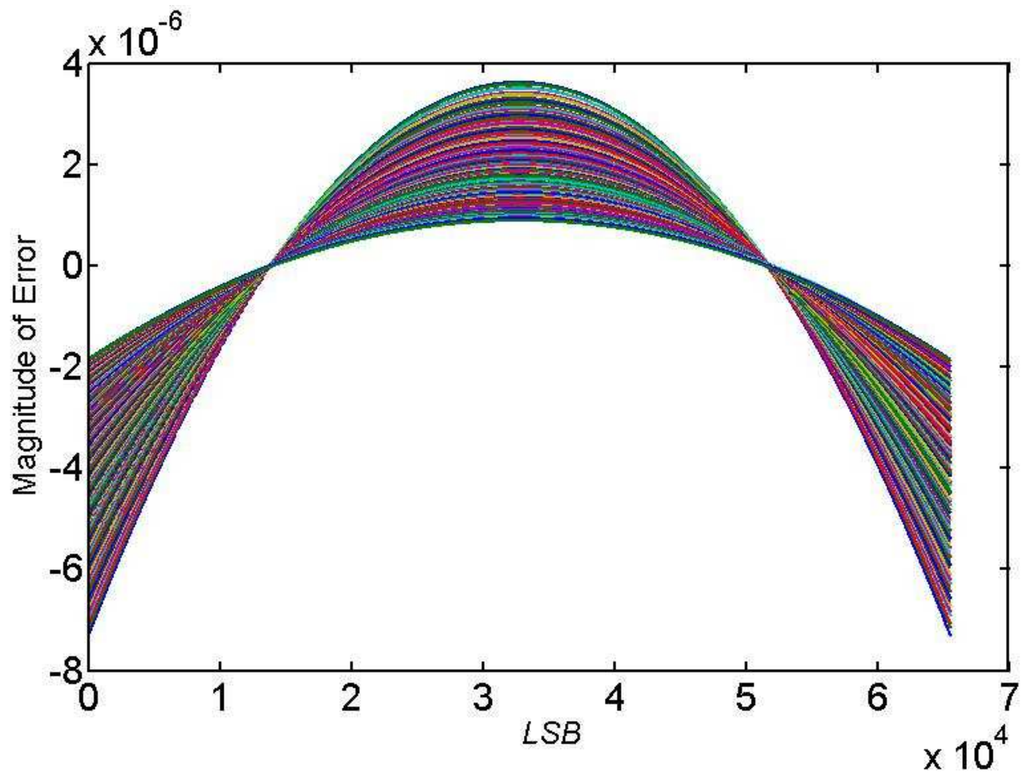


Figure C.1: Configuartion 7 : 16 :: 7 : 9 - 1st stage errors for each segment.

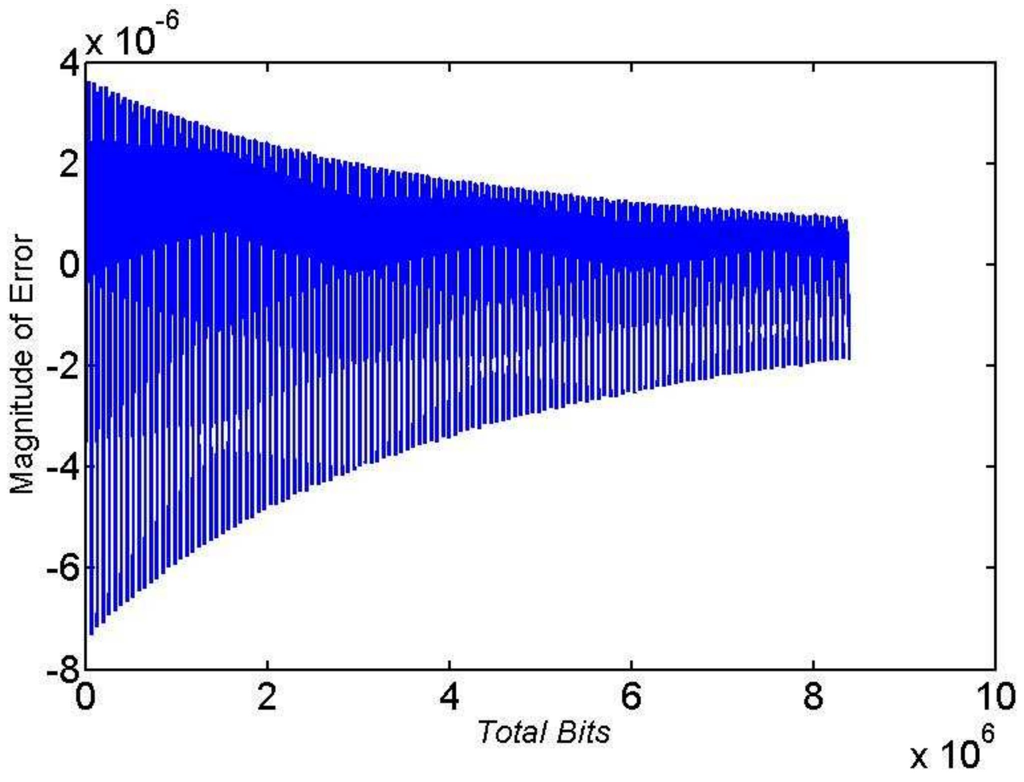


Figure C.2: Configuartion 7 : 16 :: 7 : 9 - 1st stage error approximation.

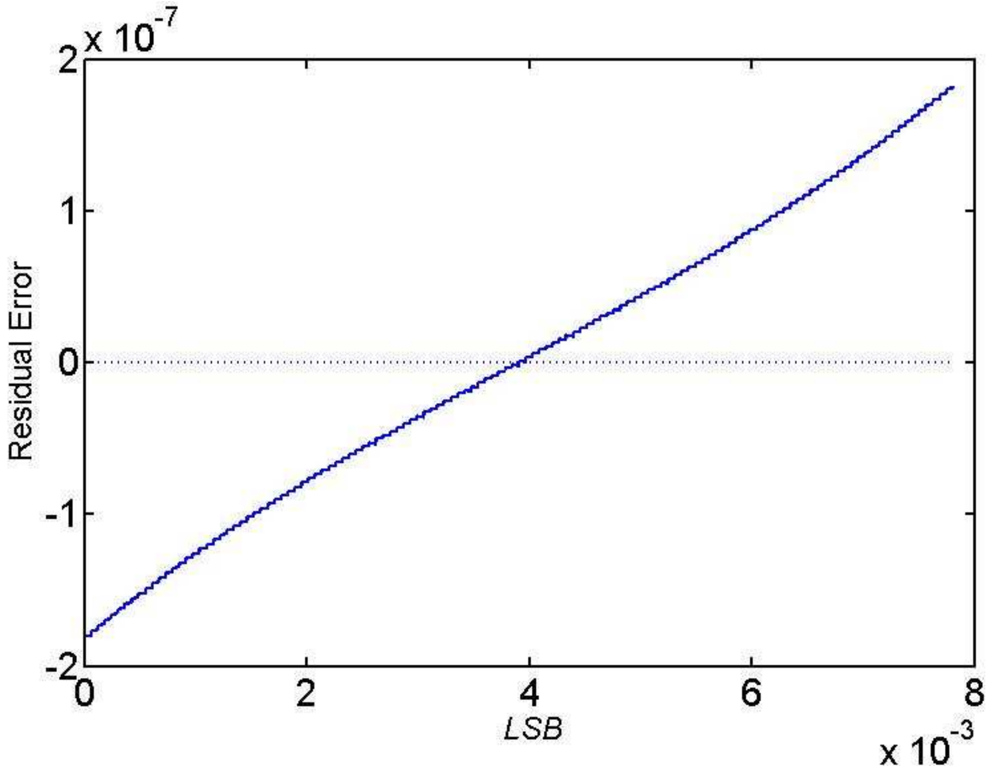


Figure C.3: Configuartion 7 : 16 :: 7 : 9 - Residual error in normalised error curve.

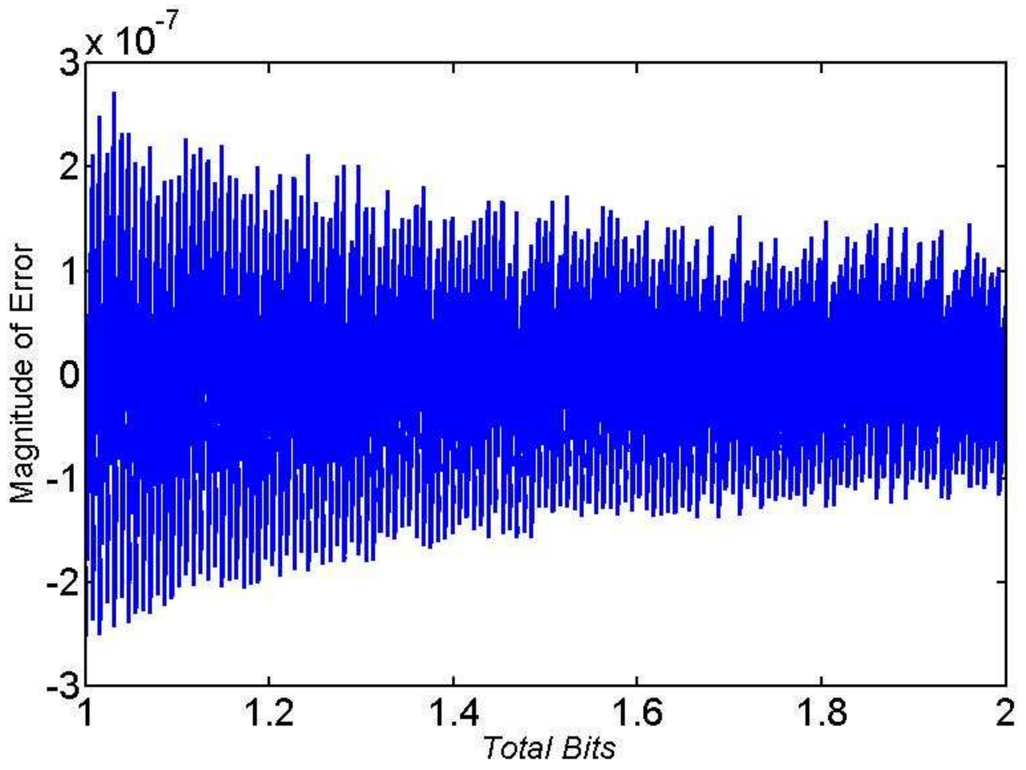


Figure C.4: Configuartion 7 : 16 :: 7 : 9 - 1st stage + 2nd stage error approximation.

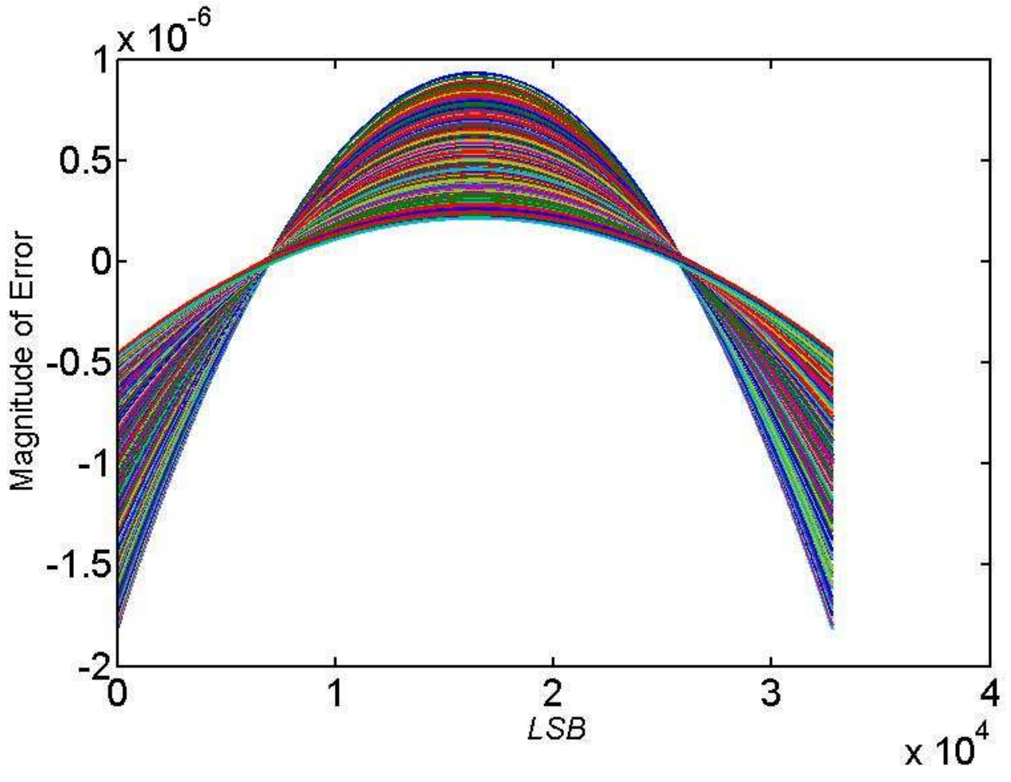


Figure C.5: Configuartion 8 : 15 :: 6 : 9 - 1st stage errors for each segment.

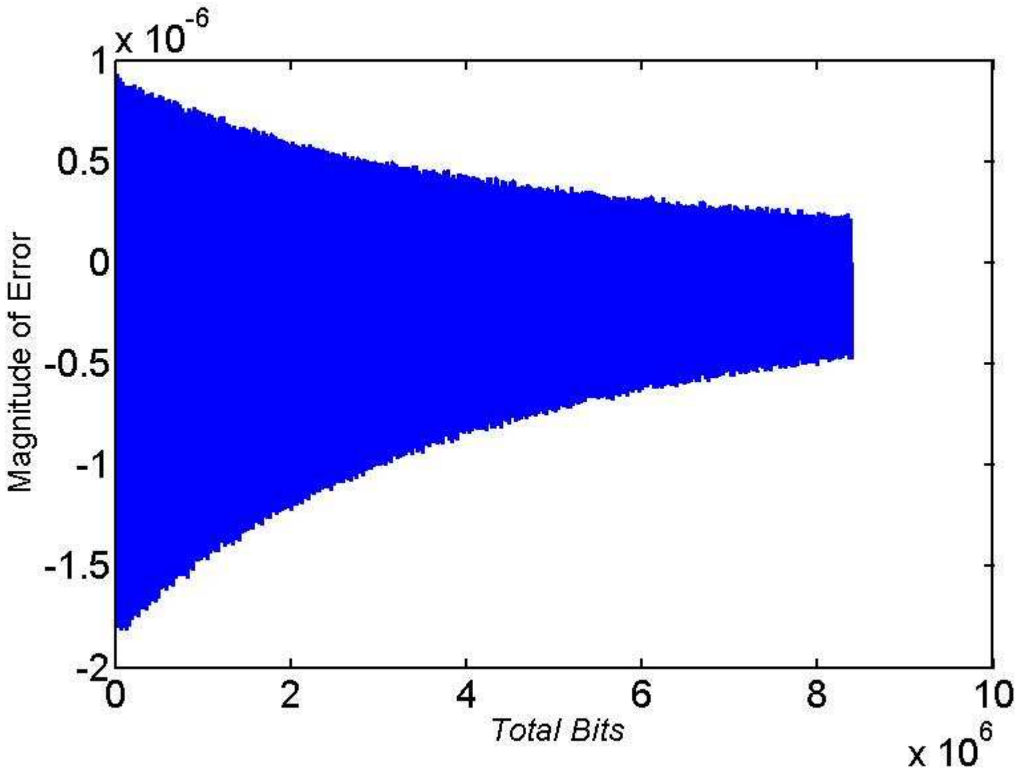


Figure C.6: Configuartion 8 : 15 :: 6 : 9 - 1st stage error approximation.

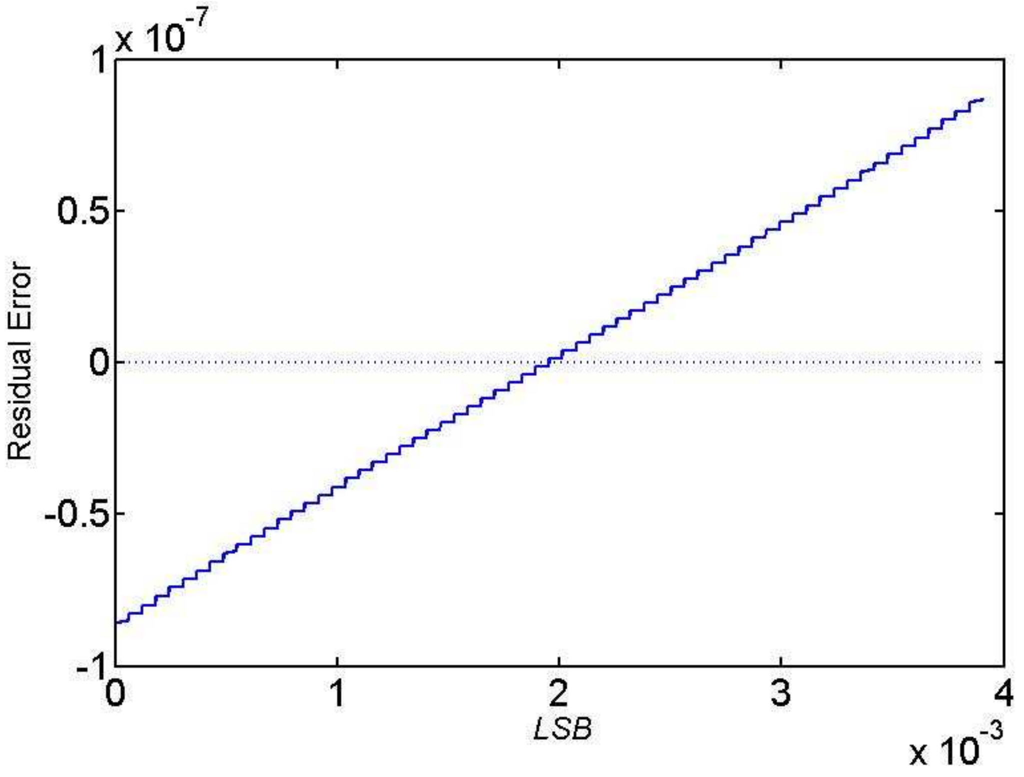


Figure C.7: Configuartion 8 : 15 :: 6 : 9 - Residual error in normalised error curve.

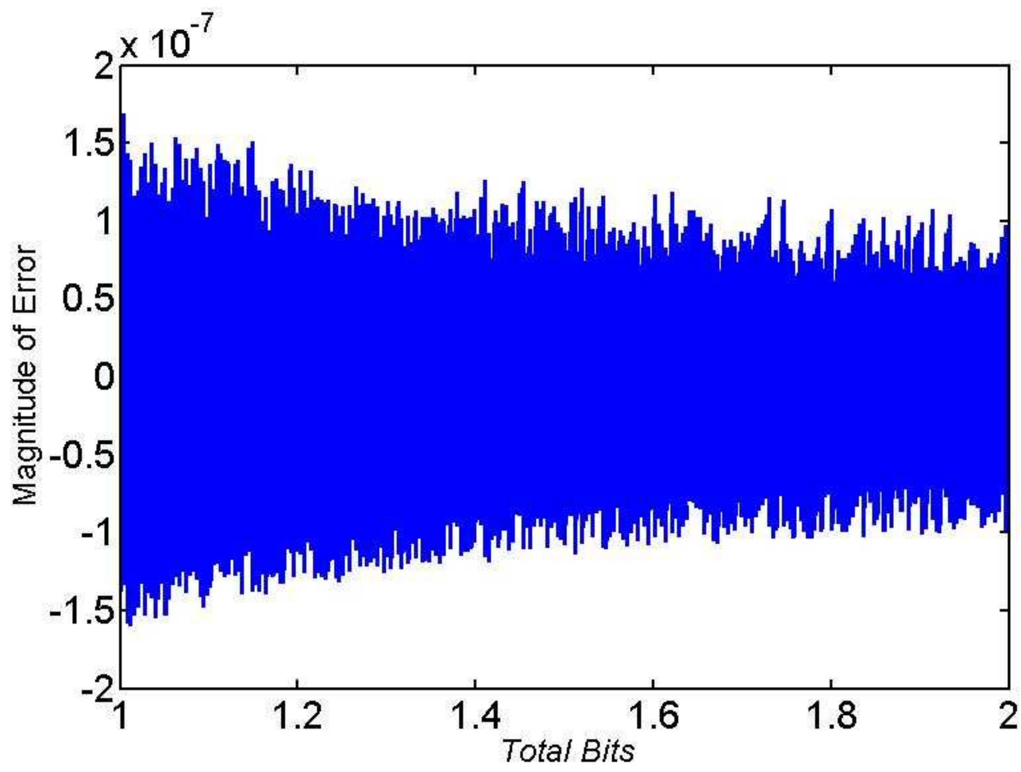


Figure C.8: Configuartion 8 : 15 :: 6 : 9 - 1st stage + 2nd stage error approximation.

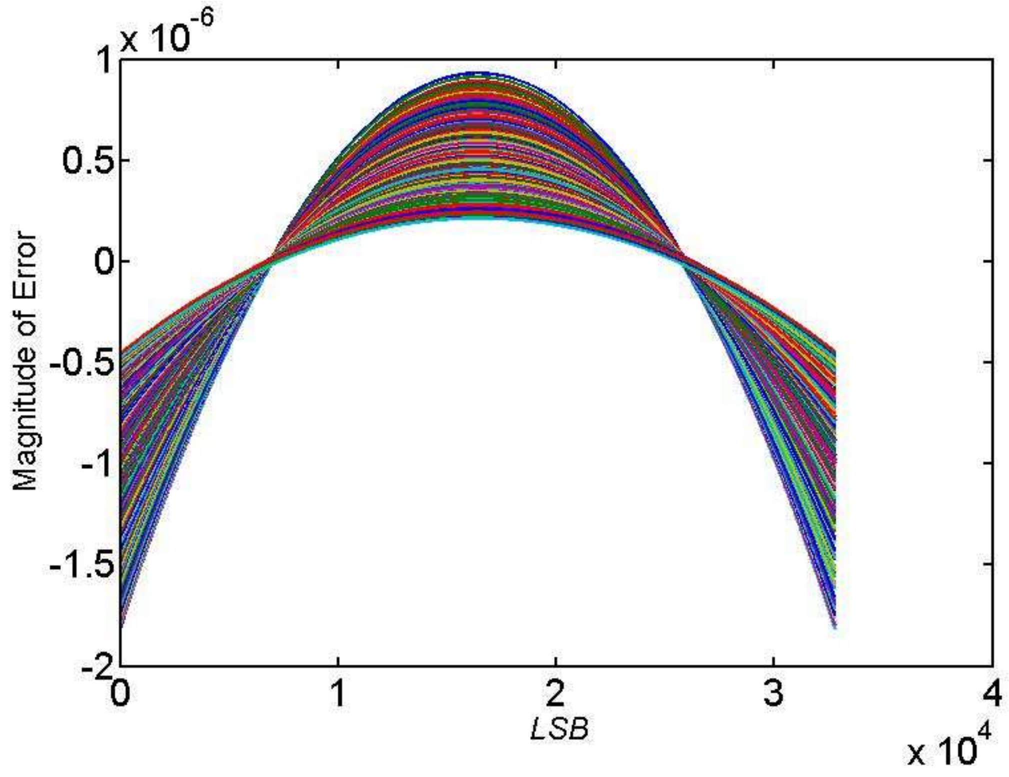


Figure C.9: Configuartion 8 : 15 :: 7 : 8 - 1st stage errors for each segment.

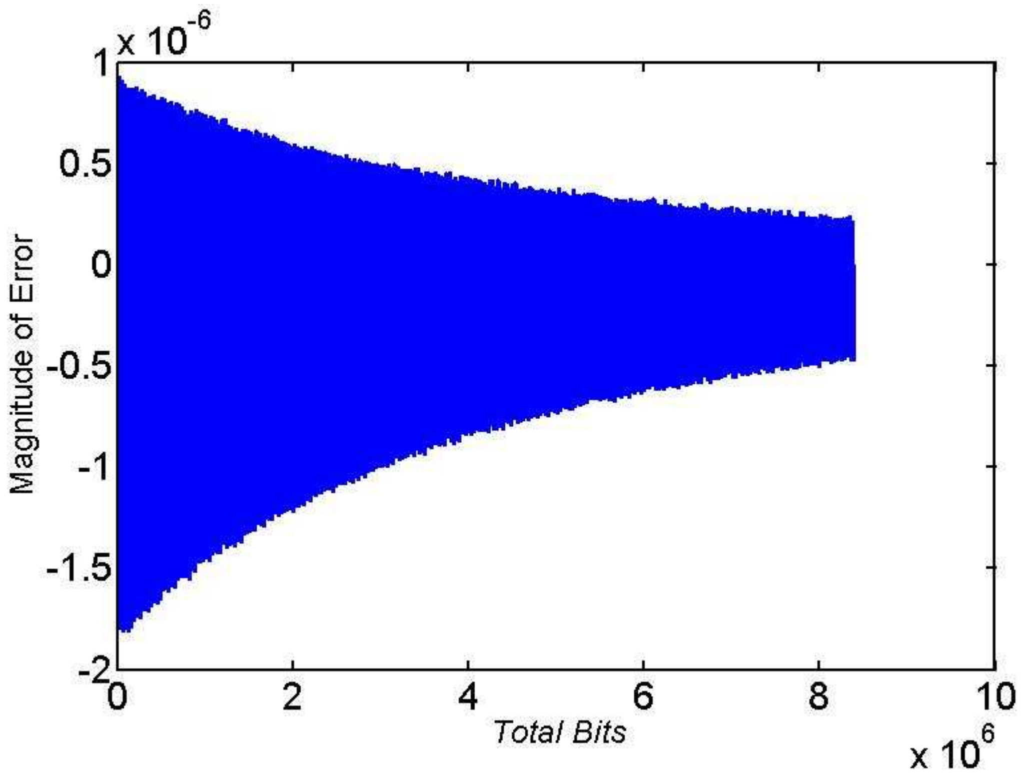


Figure C.10: Configuartion 8 : 15 :: 7 : 8 - 1st stage error approximation.

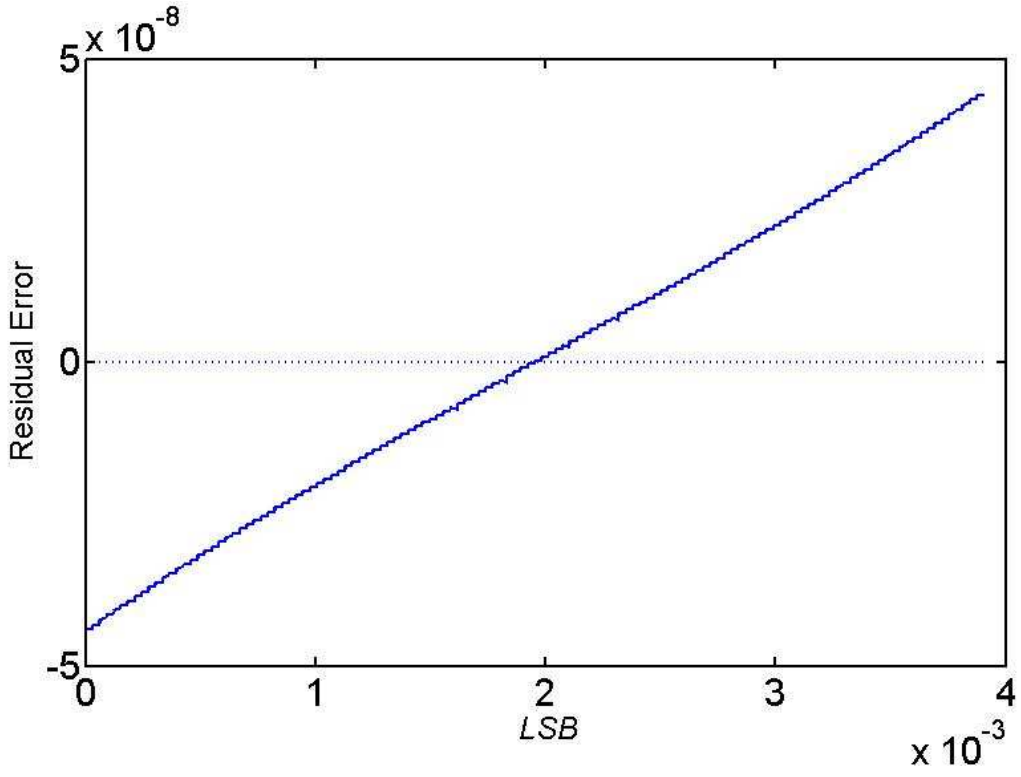


Figure C.11: Configuartion 8 : 15 :: 7 : 8 - Residual error in normalised error curve.

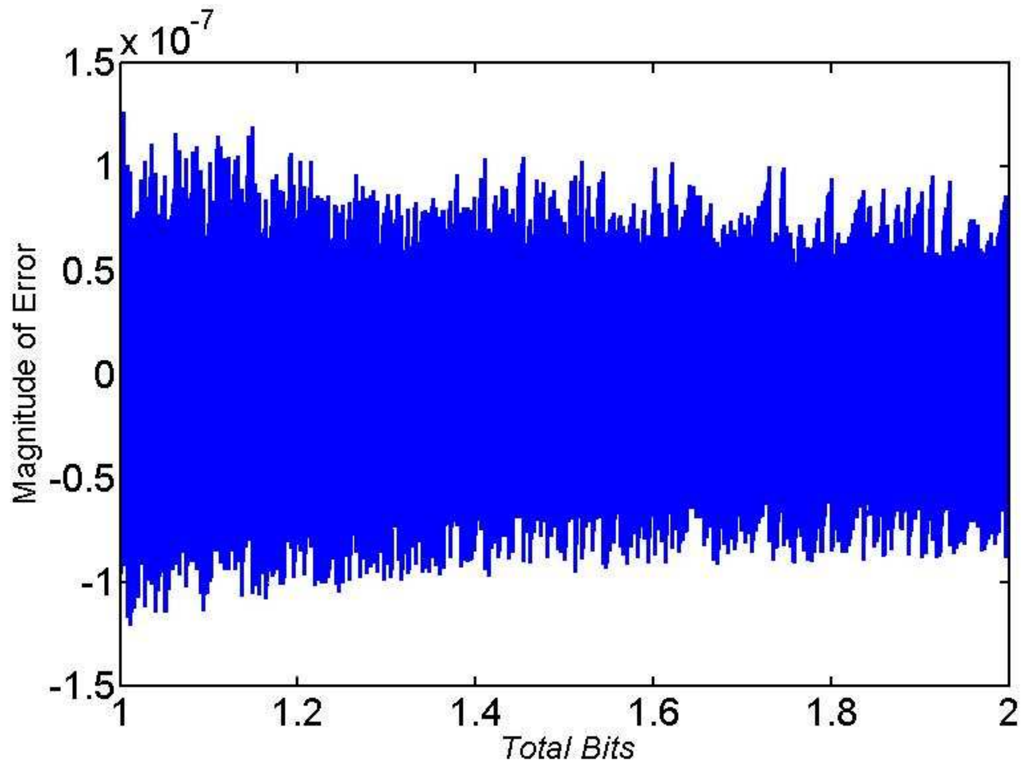


Figure C.12: Configuartion 8 : 15 :: 7 : 8 - 1st stage + 2nd stage error approximation.

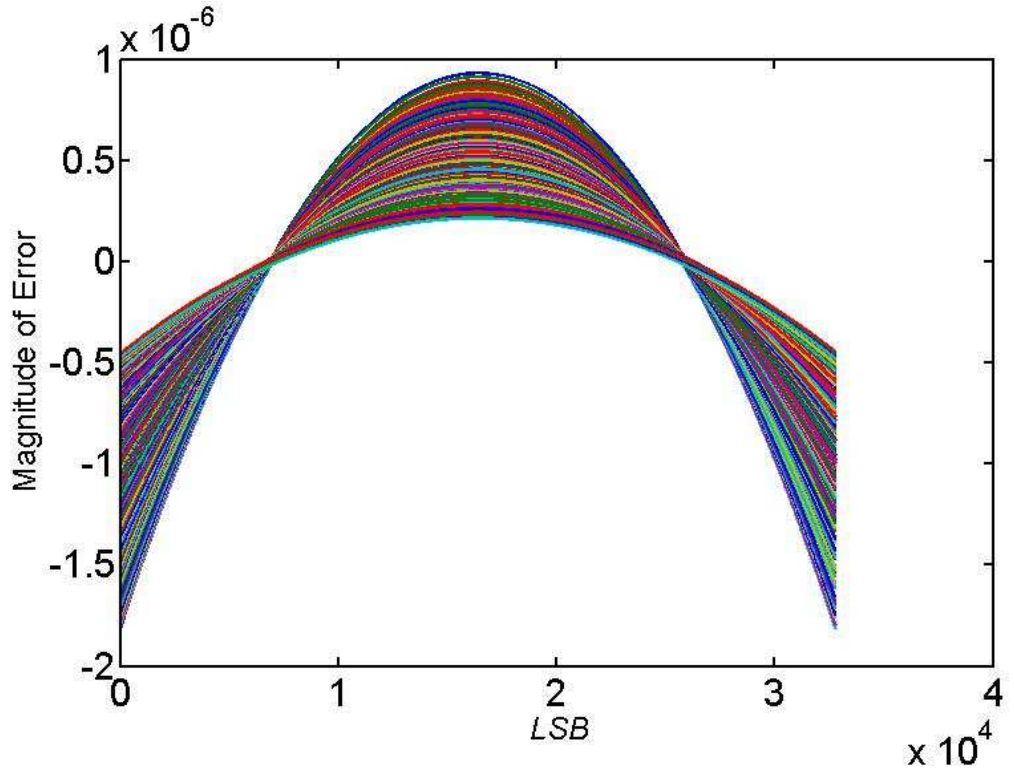


Figure C.13: Configuartion 8 : 15 :: 8 : 7 - 1st stage errors for each segment.

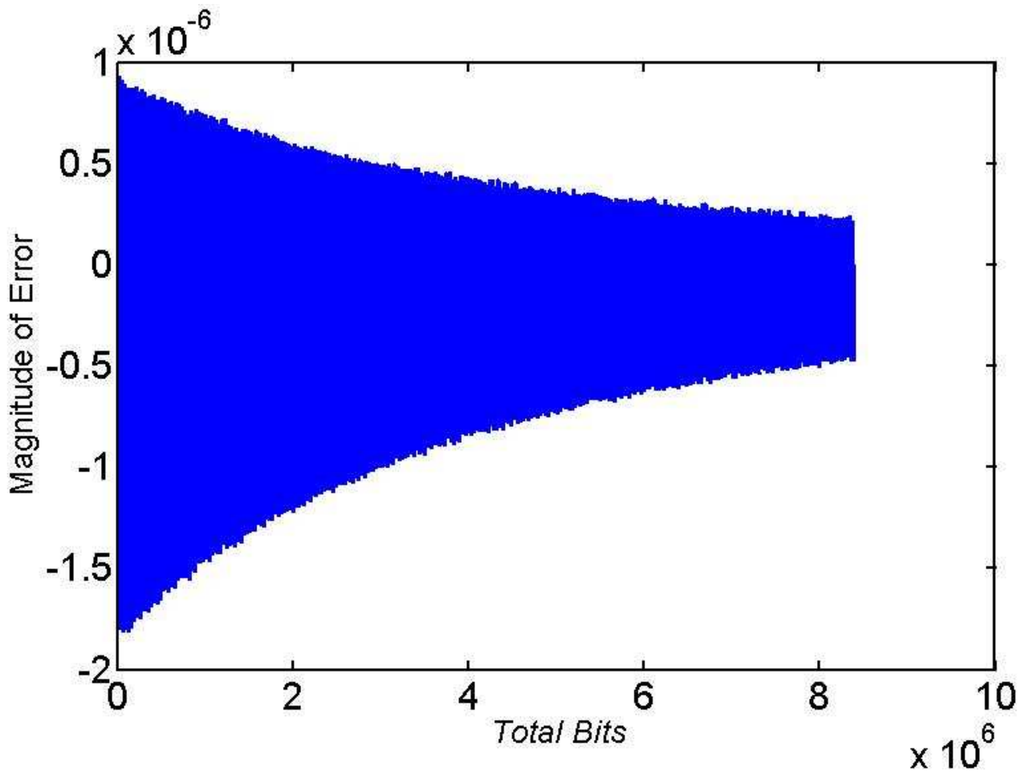


Figure C.14: Configuartion 8 : 15 :: 8 : 7 - 1st stage error approximation.

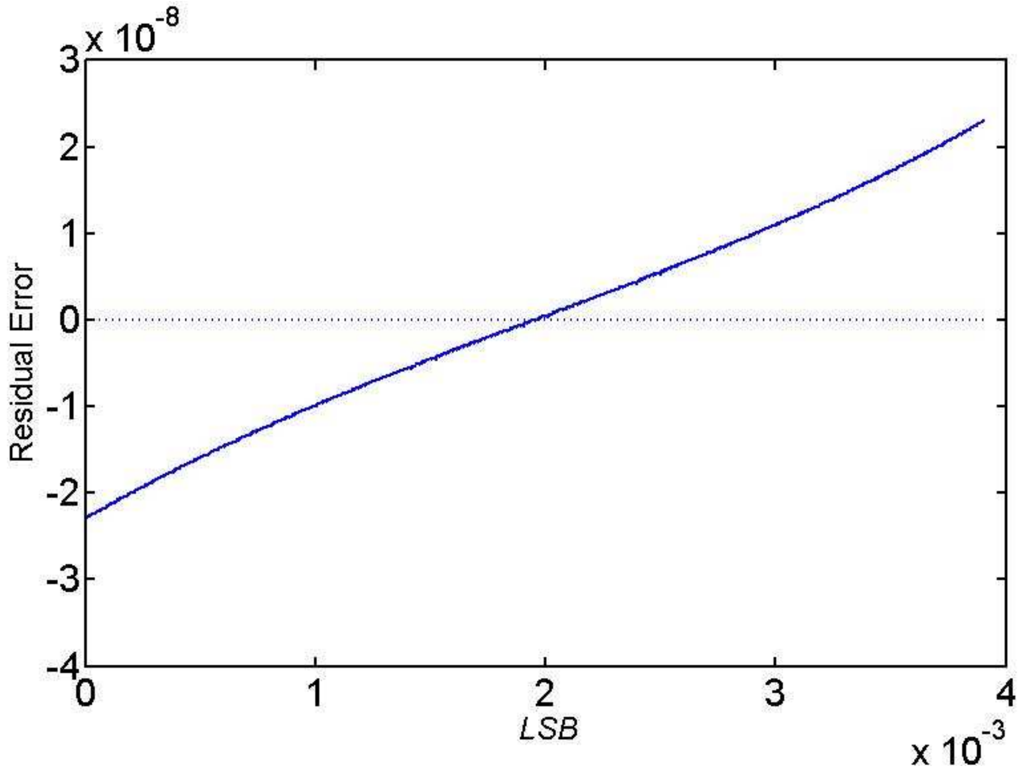


Figure C.15: Configuartion 8 : 15 :: 8 : 7 - Residual error in normalised error curve.

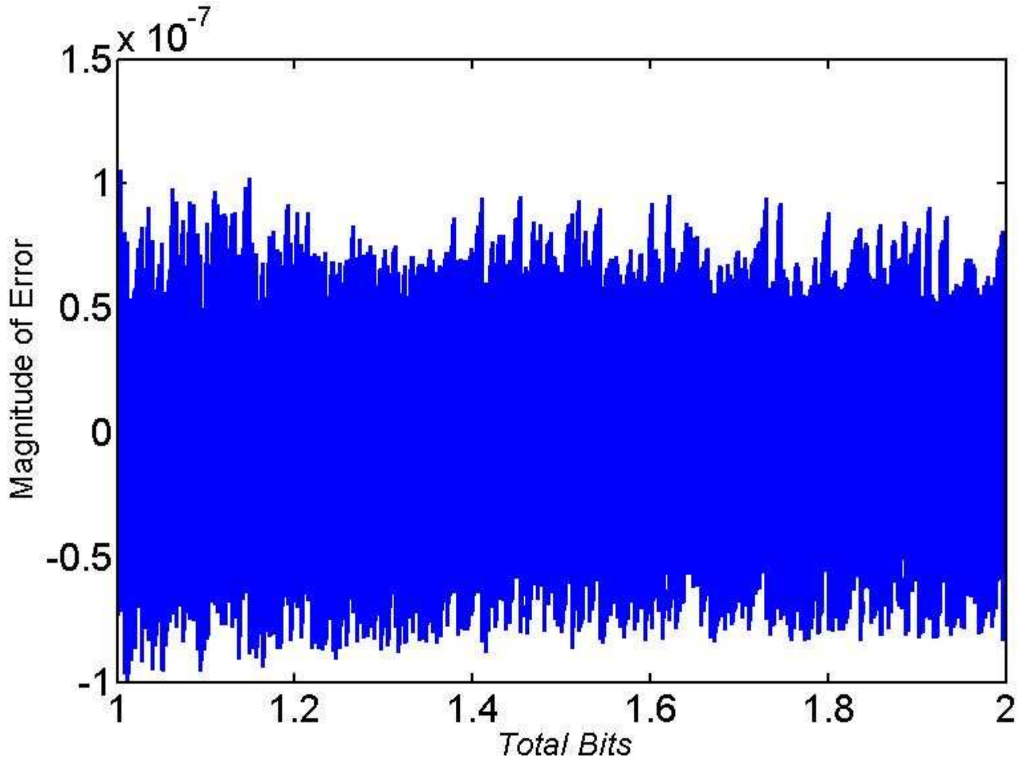


Figure C.16: Configuartion 8 : 15 :: 8 : 7 - 1st stage + 2nd stage error approximation.

# Appendix D

# Table 5.3 Matlab Simulations

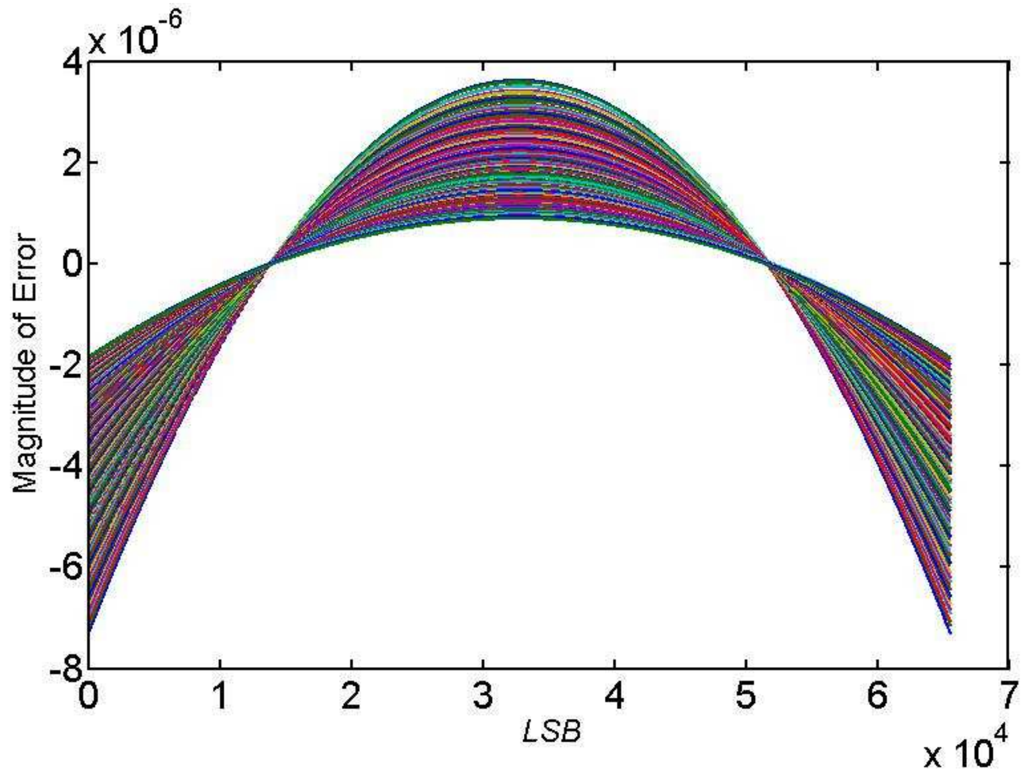


Figure D.1: Configuartion 7 : 16 :: 7 : 9 - 1st stage errors for each segment.

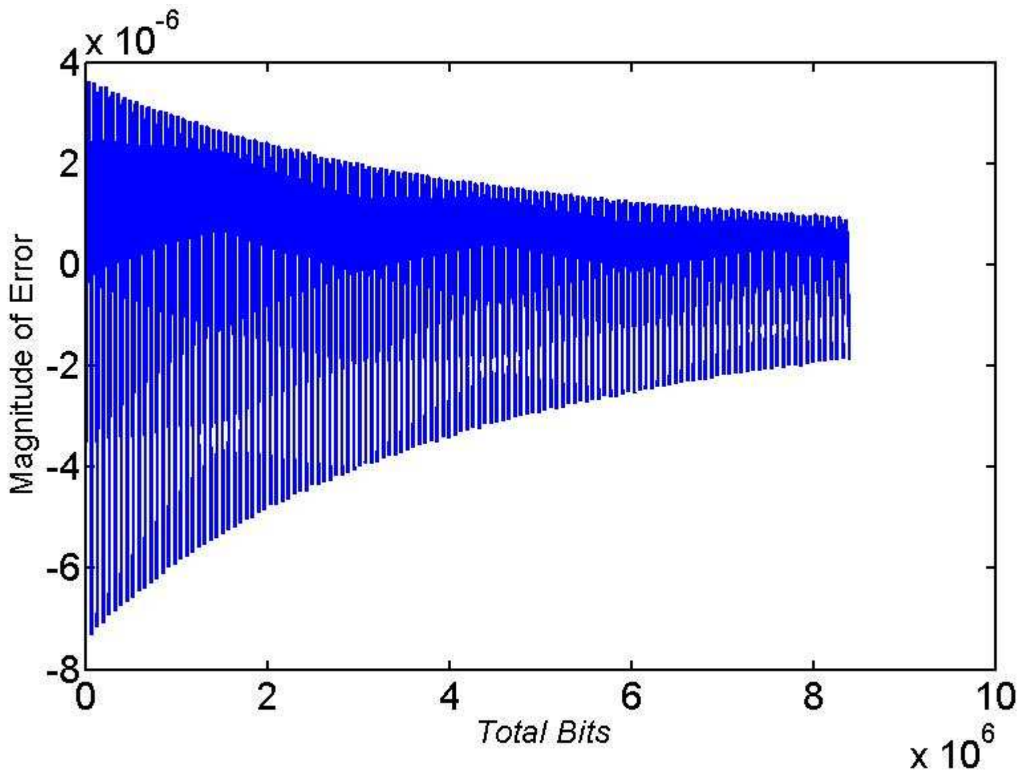


Figure D.2: Configuartion 7 : 16 :: 7 : 9 - 1st stage error approximation.

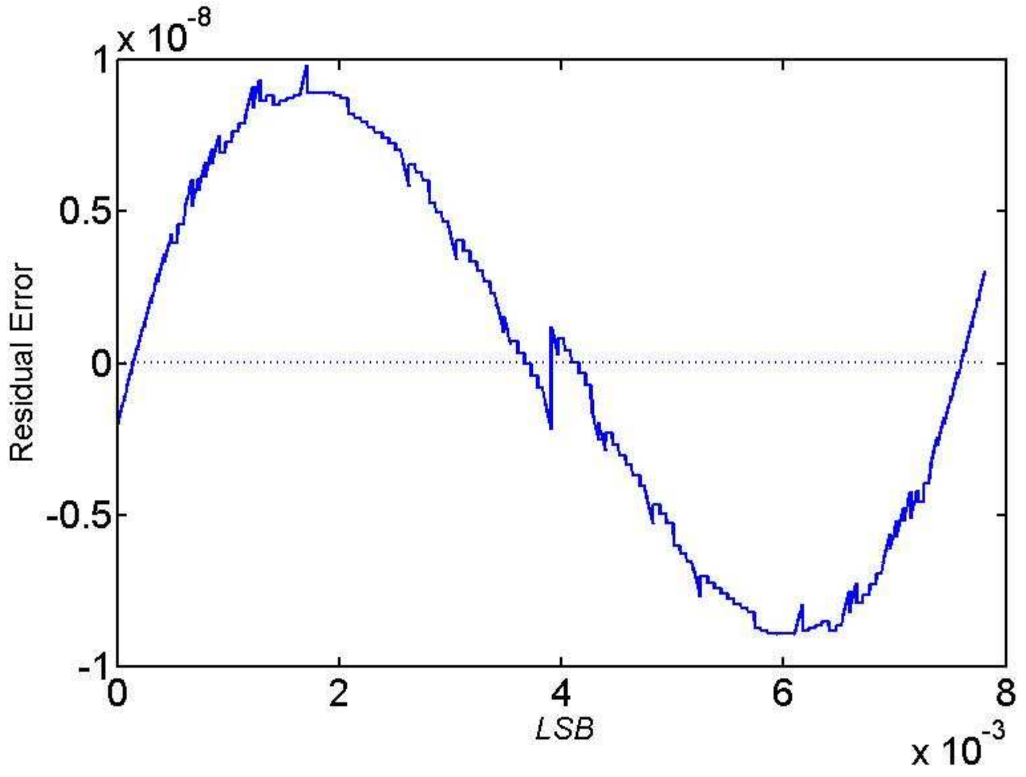


Figure D.3: Configuartion 7 : 16 :: 7 : 9 - Residual error in normalised error curve.

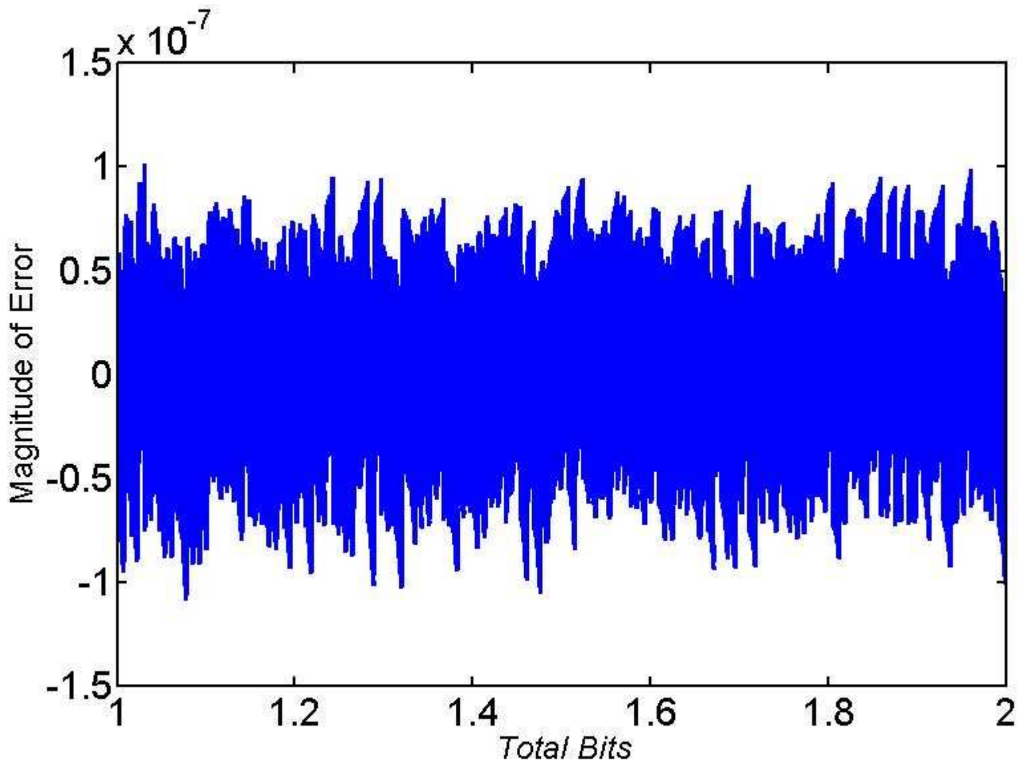


Figure D.4: Configuartion 7 : 16 :: 7 : 9 - 1st stage + 2nd stage error approximation.

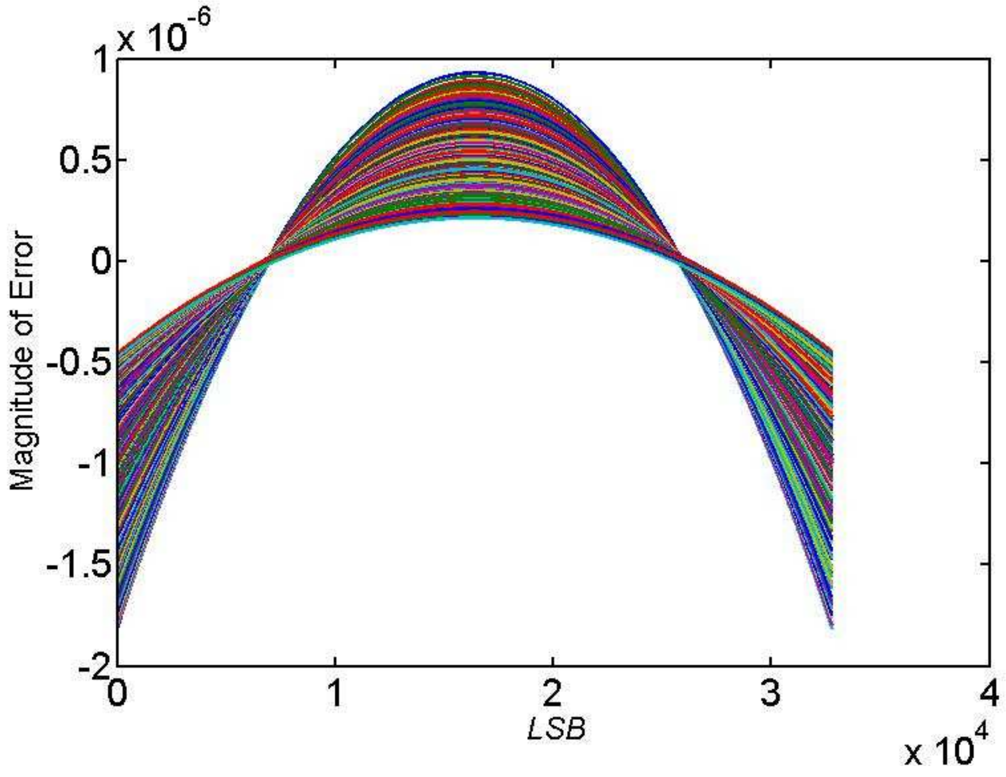


Figure D.5: Configuartion 8 : 15 :: 6 : 9 - 1st stage errors for each segment.

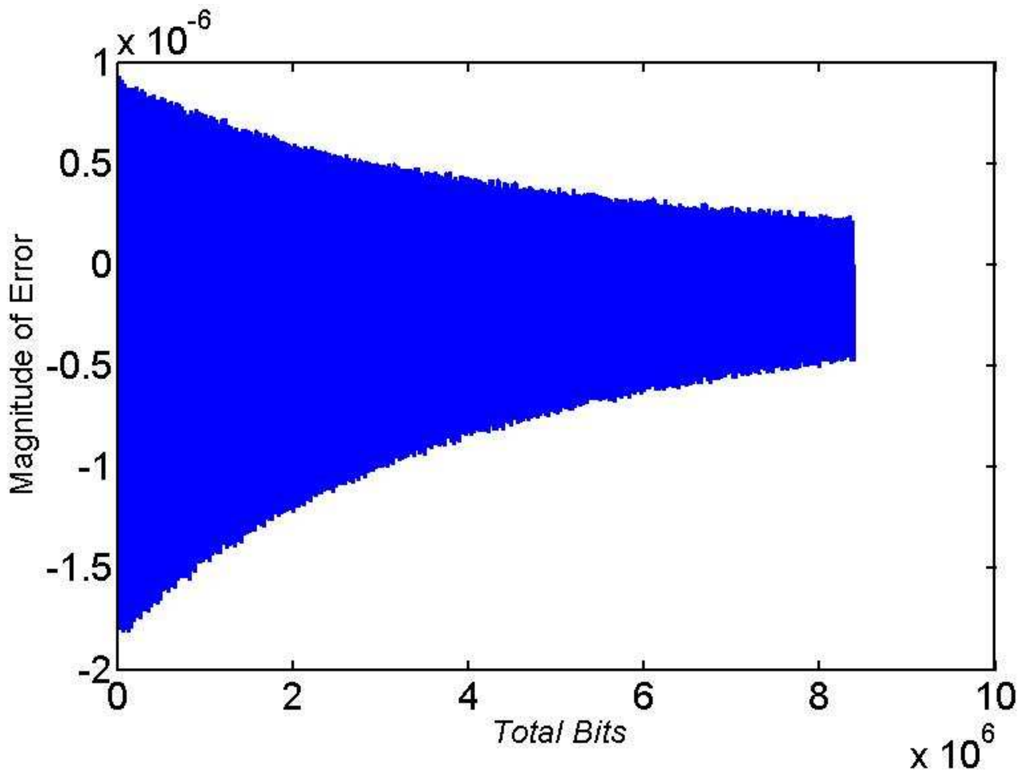


Figure D.6: Configuartion 8 : 15 :: 6 : 9 - 1st stage error approximation.

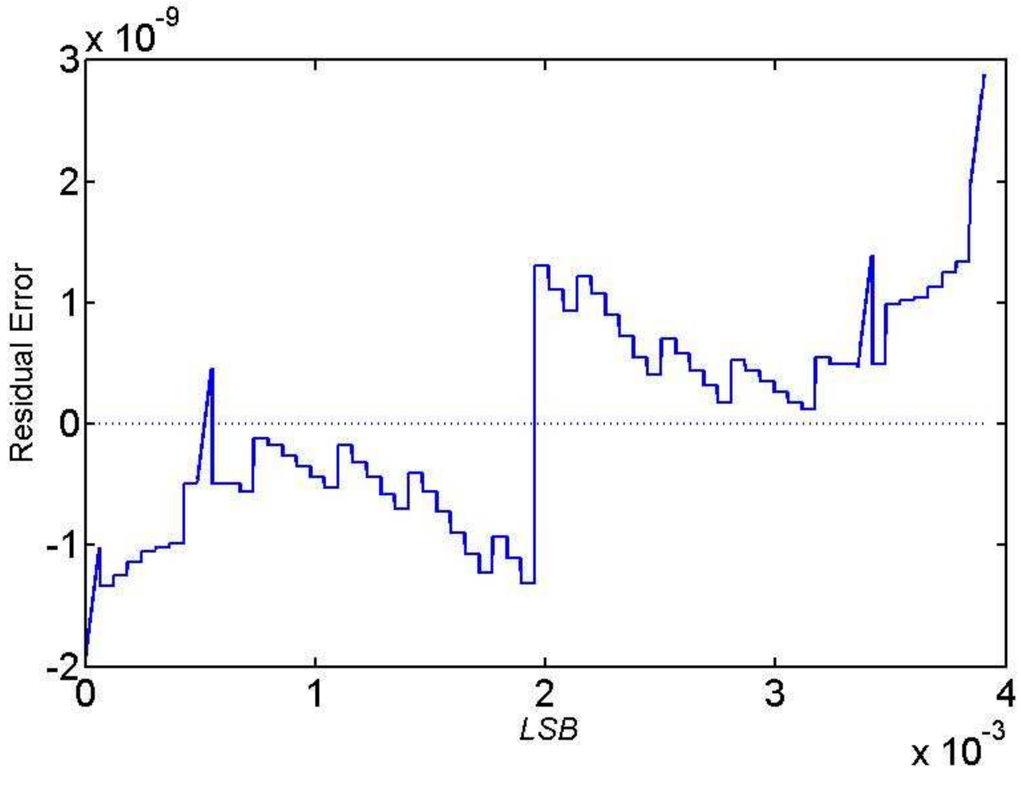


Figure D.7: Configuartion 8 : 15 :: 6 : 9 - Residual error in normalised error curve.

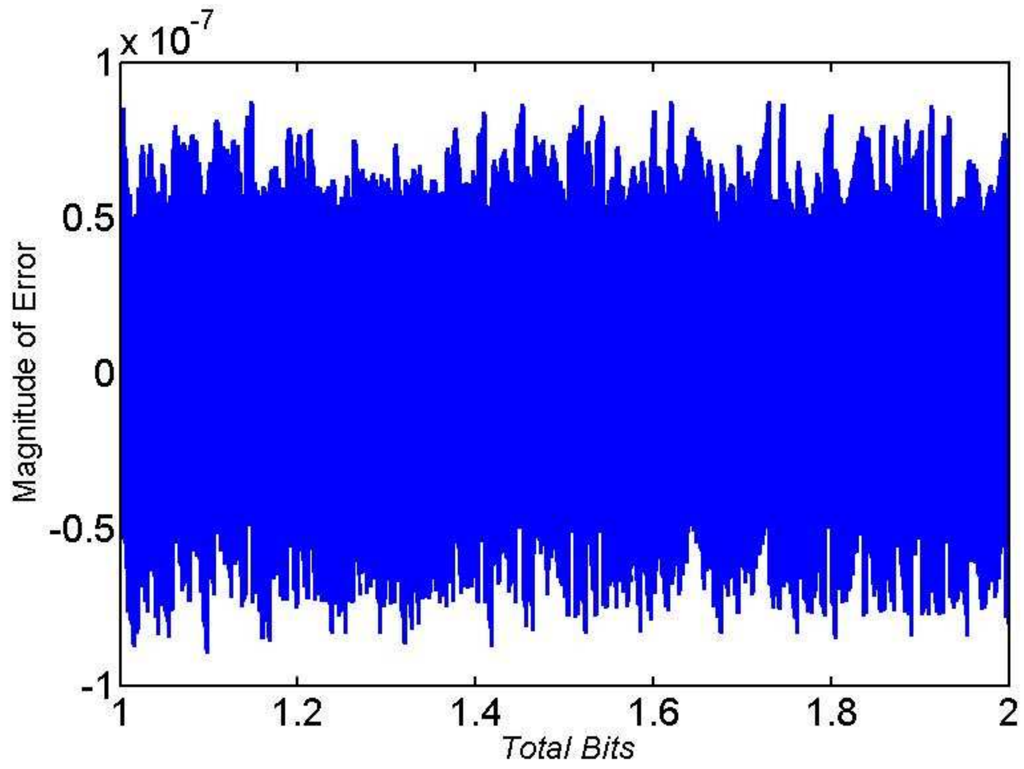


Figure D.8: Configuartion 8 : 15 :: 6 : 9 - 1st stage + 2nd stage error approximation.

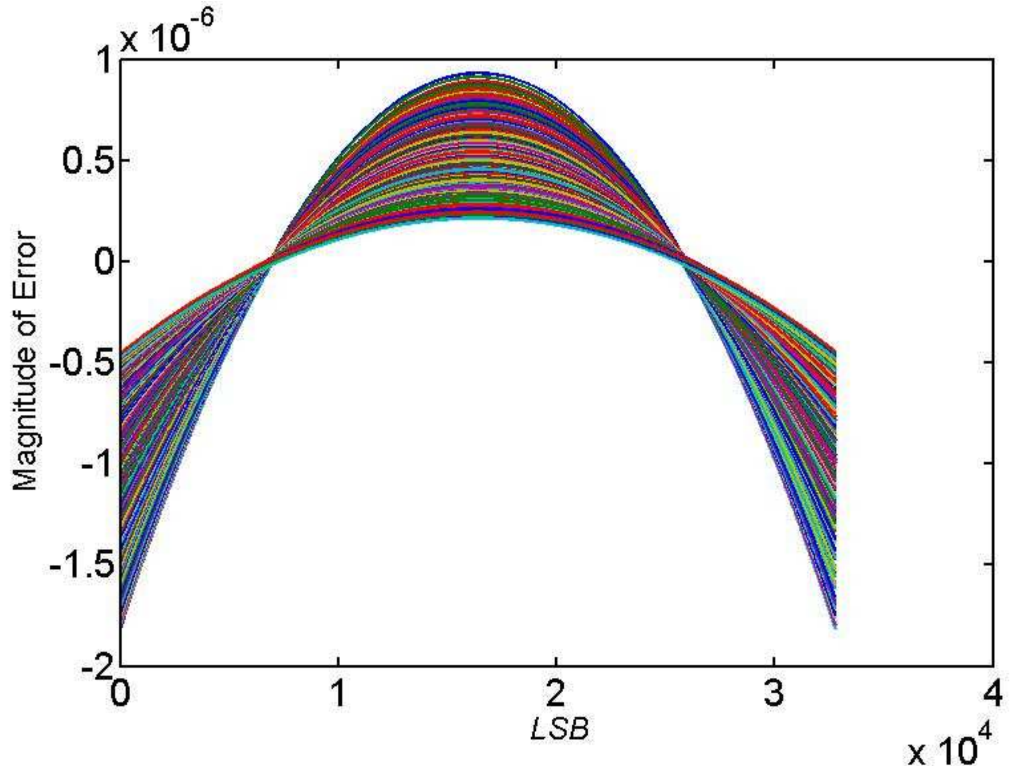


Figure D.9: Configuartion 8 : 15 :: 7 : 8 - 1st stage errors for each segment.

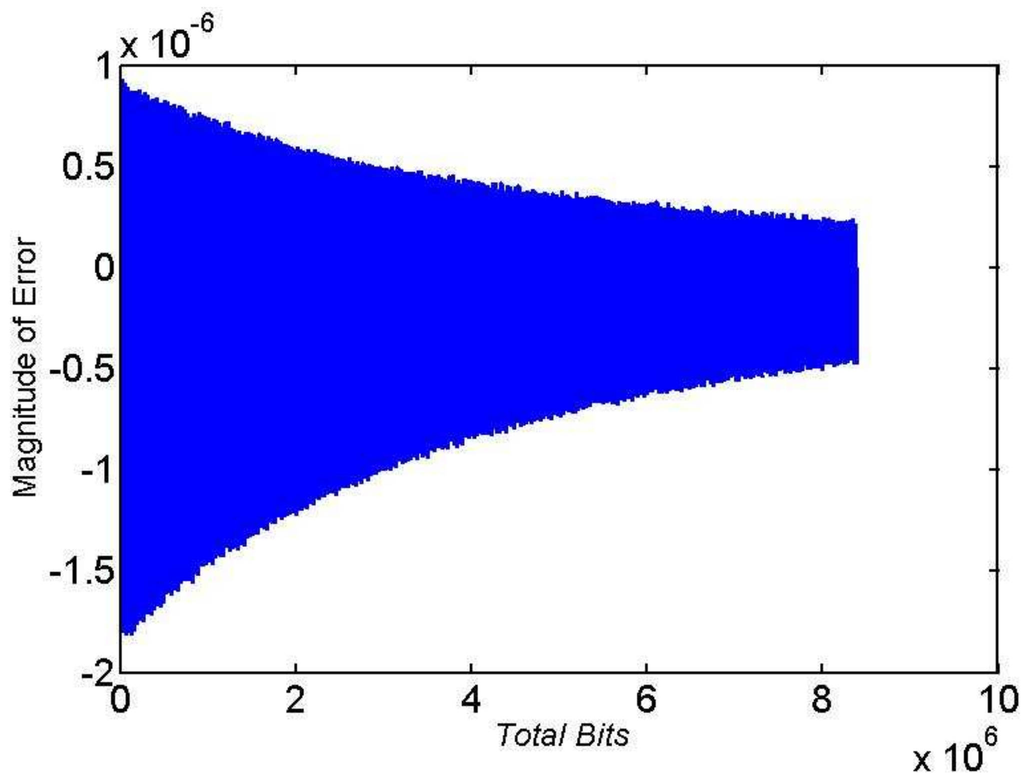


Figure D.10: Configuartion 8 : 15 :: 7 : 8 - 1st stage error approximation.

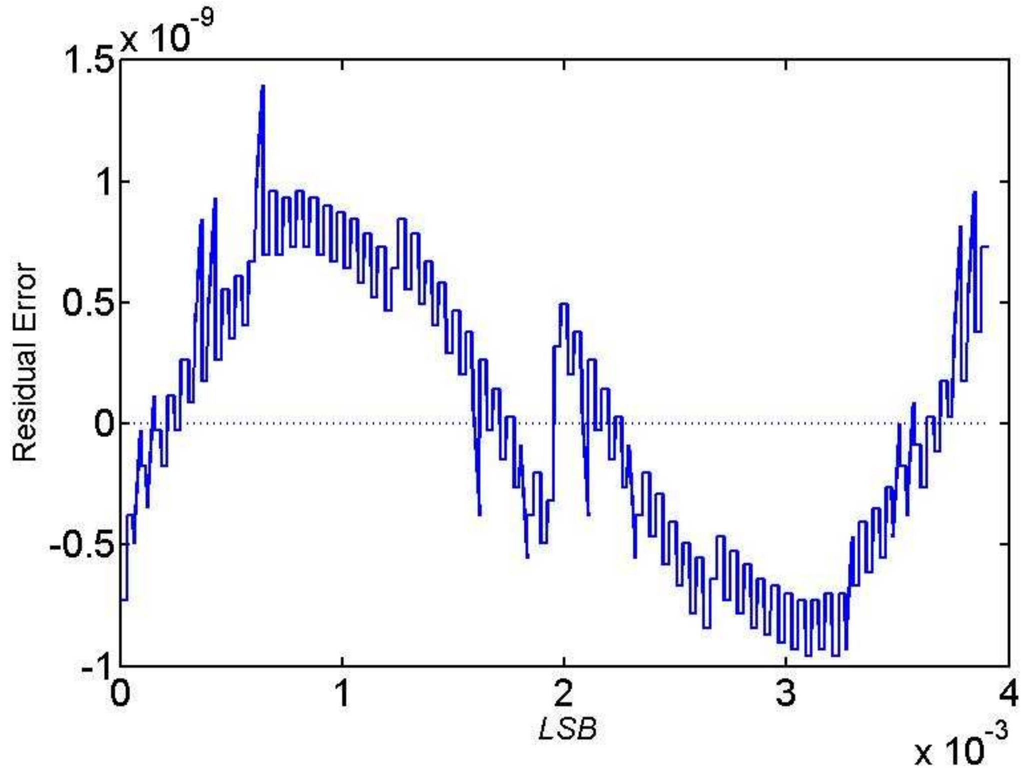


Figure D.11: Configuartion 8 : 15 :: 7 : 8 - Residual error in normalised error curve.

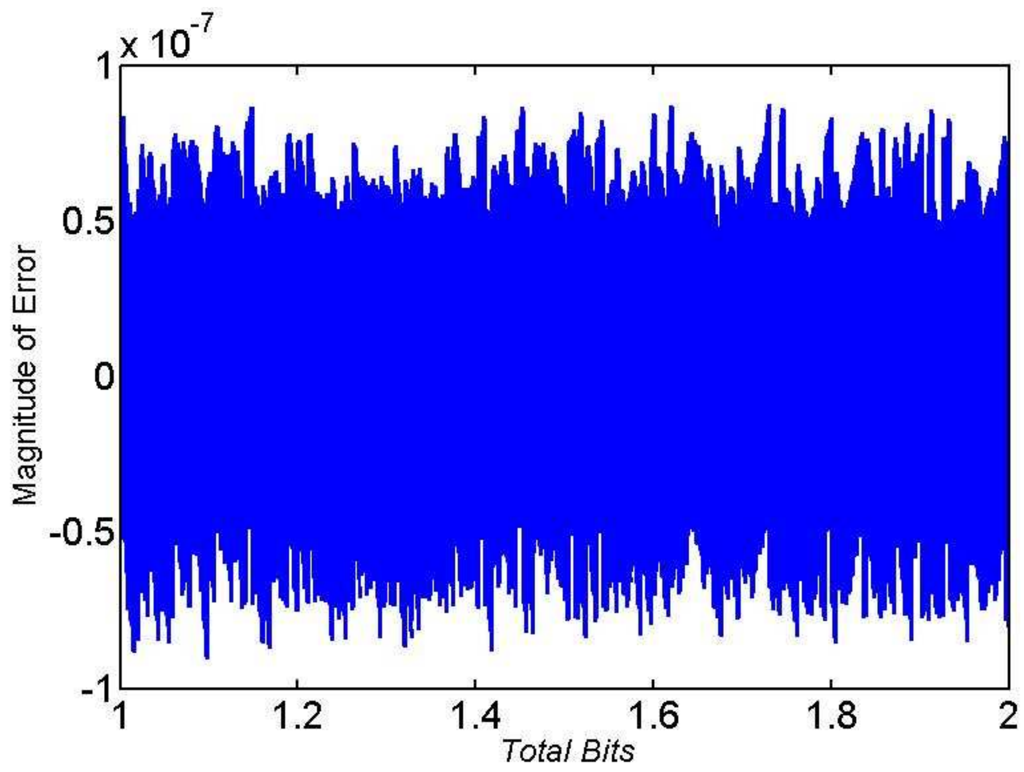


Figure D.12: Configuartion 8 : 15 :: 7 : 8 - 1st stage + 2nd stage error approximation.

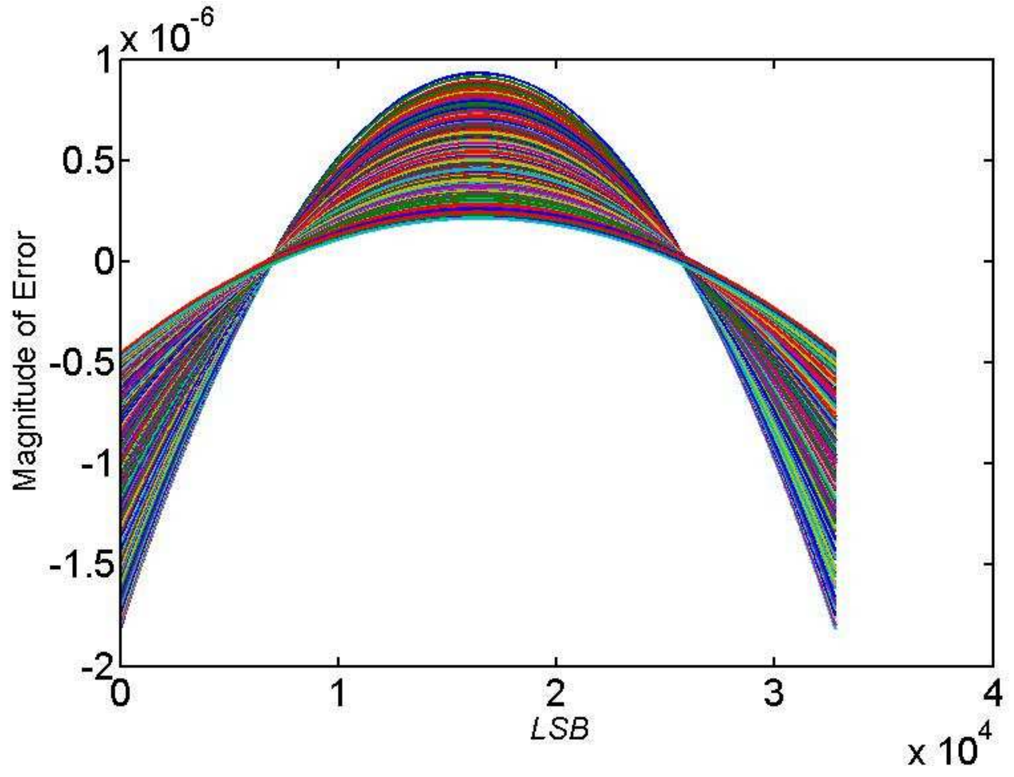


Figure D.13: Configuartion 8 : 15 :: 8 : 7 - 1st stage errors for each segment.

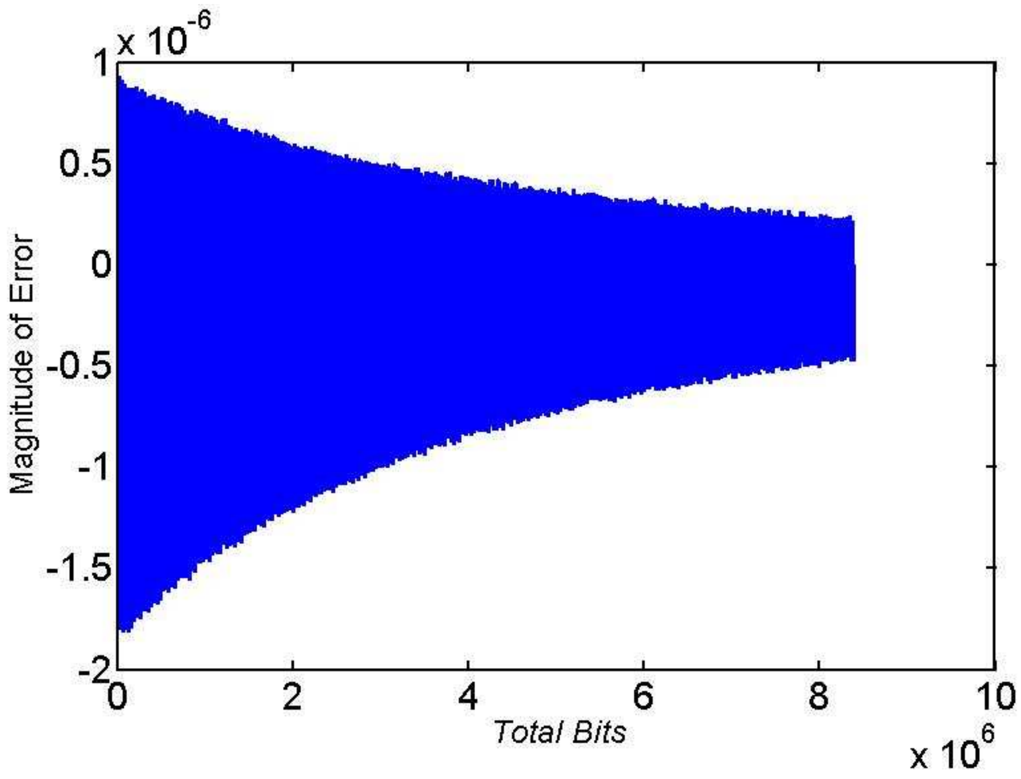


Figure D.14: Configuartion 8 : 15 :: 8 : 7 - 1st stage error approximation.

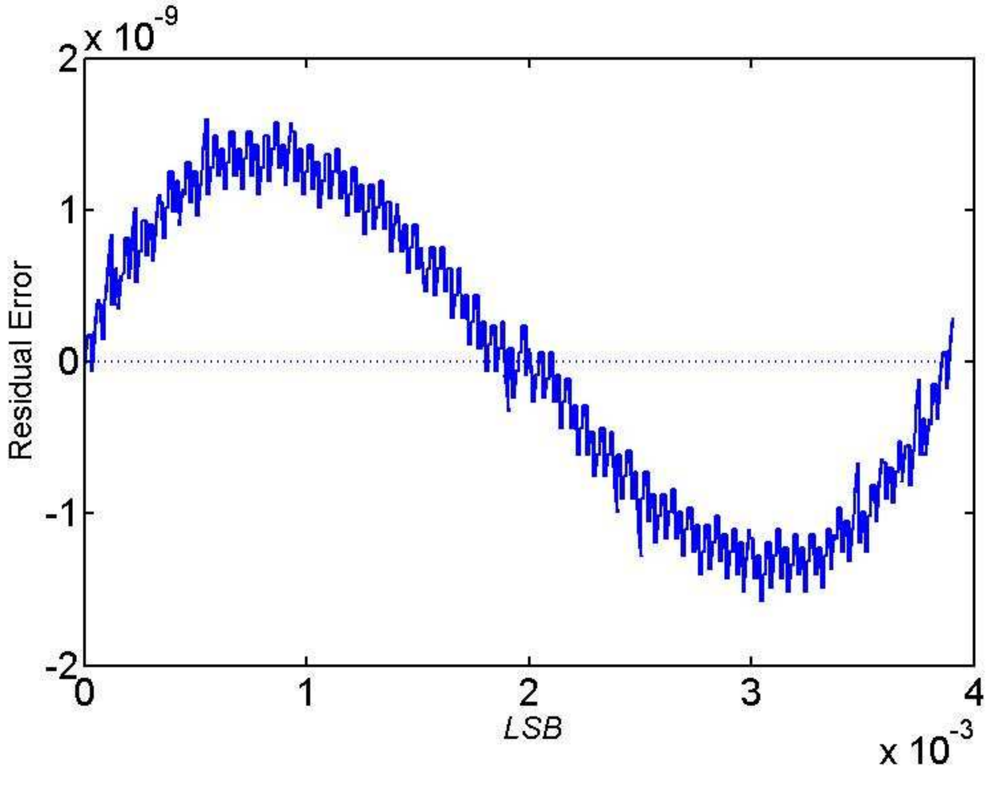


Figure D.15: Configuartion 8 : 15 :: 8 : 7 - Residual error in normalised error curve.

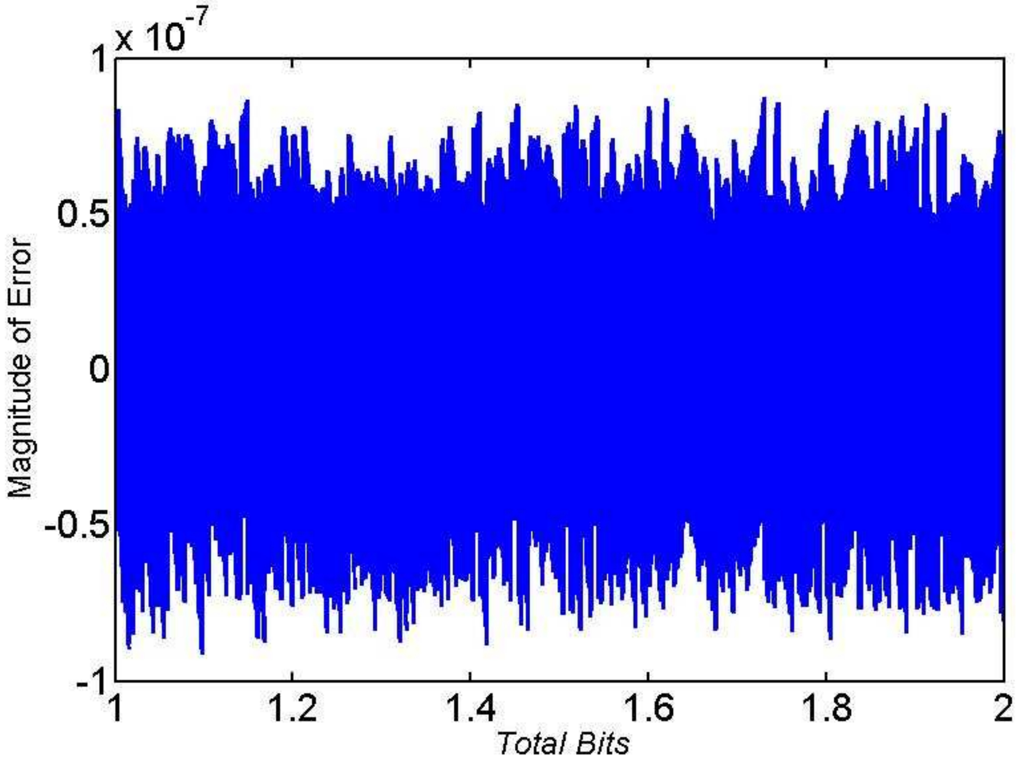


Figure D.16: Configuartion 8 : 15 :: 8 : 7 - 1st stage + 2nd stage error approximation.

# Appendix E

# Logarithmic Domain Cross-Correlation Plots

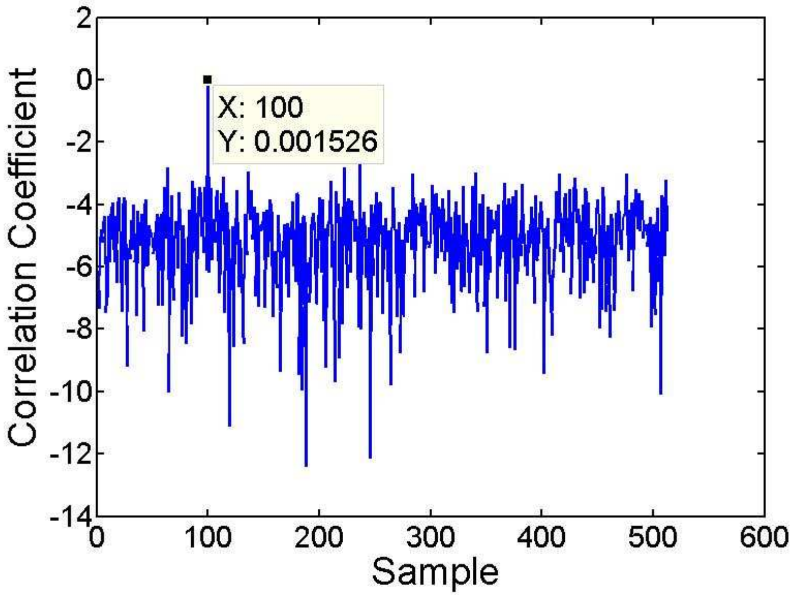


Figure E.1: Cross correlation coefficient value of first combination - using 16 fractional bits.

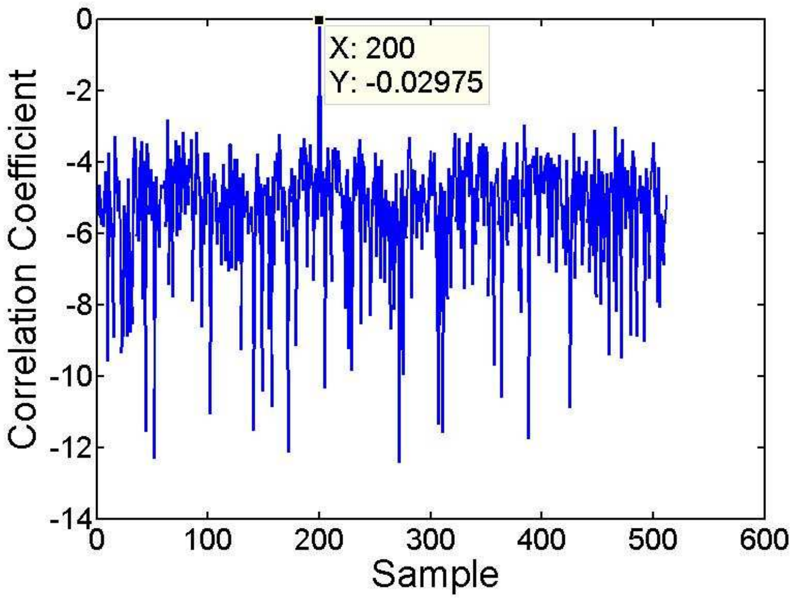


Figure E.2: Cross correlation coefficient value of second combination - using 16 fractional bits.

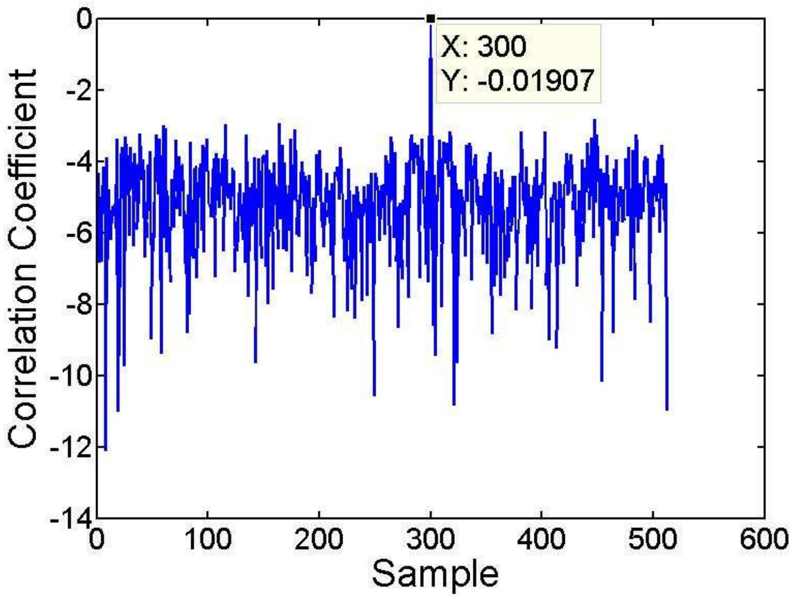


Figure E.3: Cross correlation coefficient value of third combination - using 16 fractional bits.

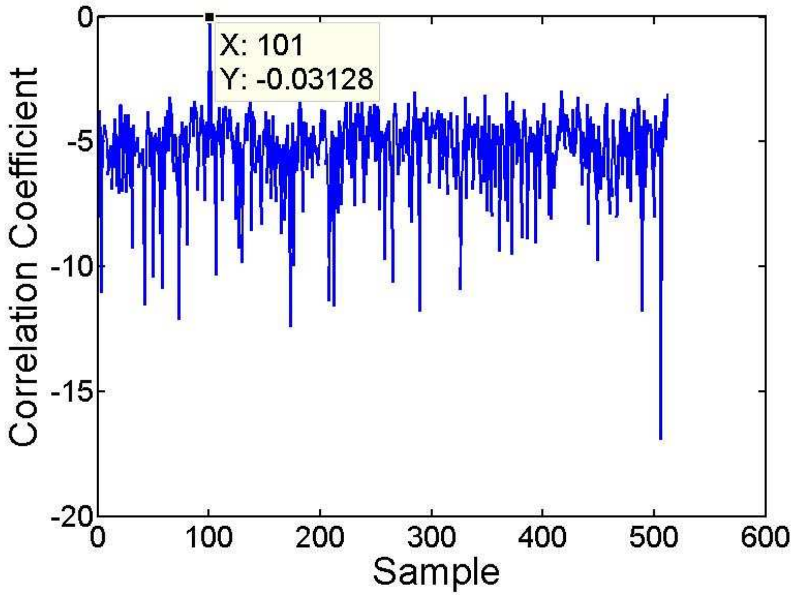


Figure E.4: Cross correlation coefficient value of fourth combination - using 16 fractional bits.

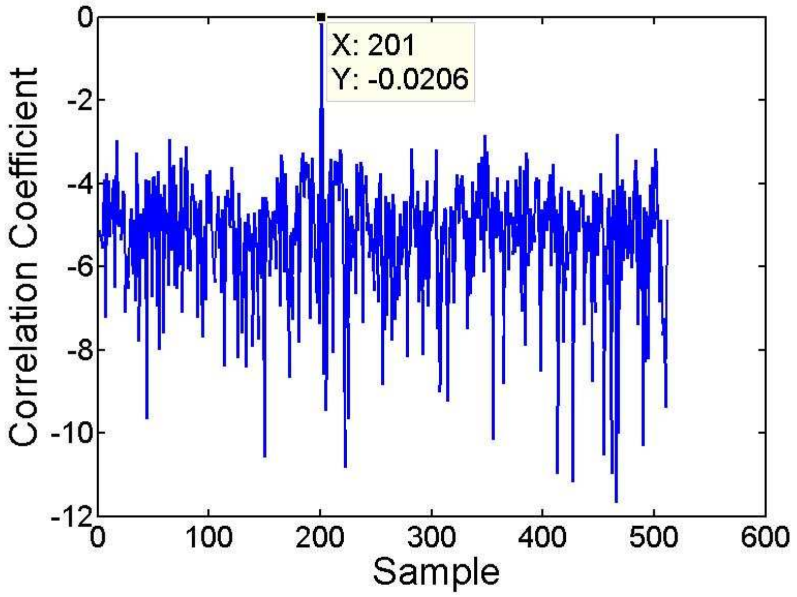


Figure E.5: Cross correlation coefficient value of fifth combination - using 16 fractional bits.

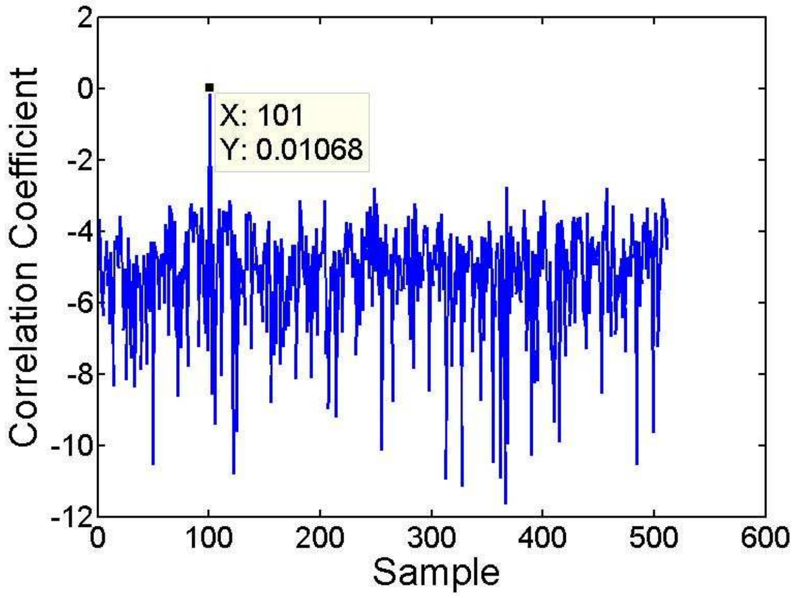


Figure E.6: Cross correlation coefficient value of sixth combination - using 16 fractional bits.

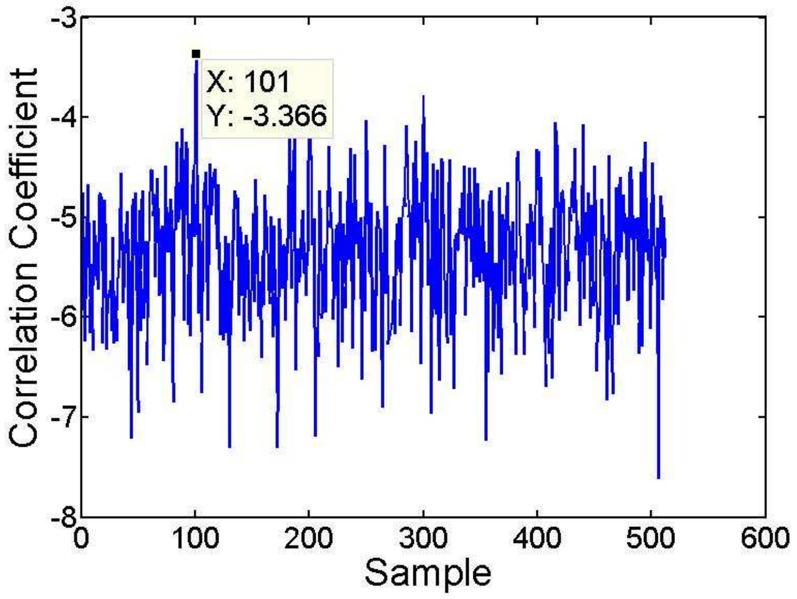


Figure E.7: Log Approximation Based Simple Moving Average of Six combinations - using 16 fractional bits.

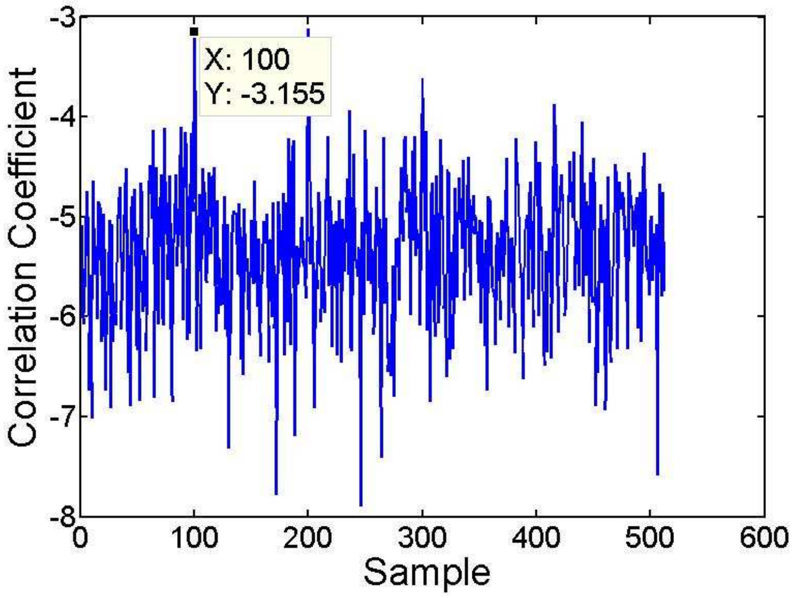


Figure E.8: Log Approximation Based Weighted Moving Average of Six combinations - using 16 fractional bits.

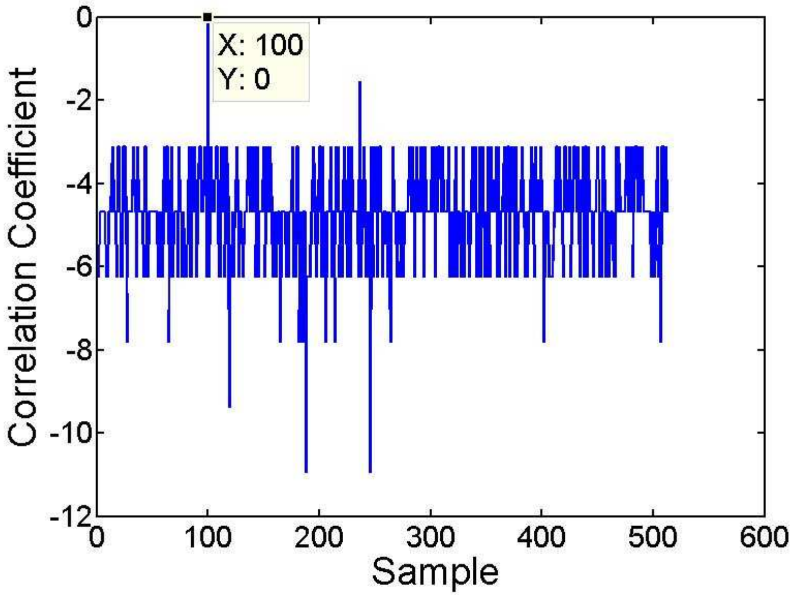


Figure E.9: Cross correlation coefficient value of first combination - using 4 fractional bits.

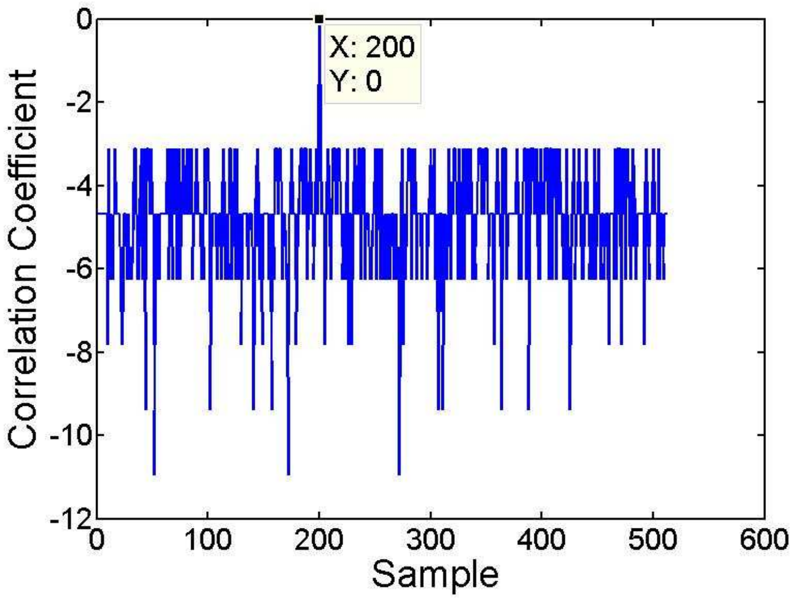


Figure E.10: Cross correlation coefficient value of second combination - using 6 fractional bits.

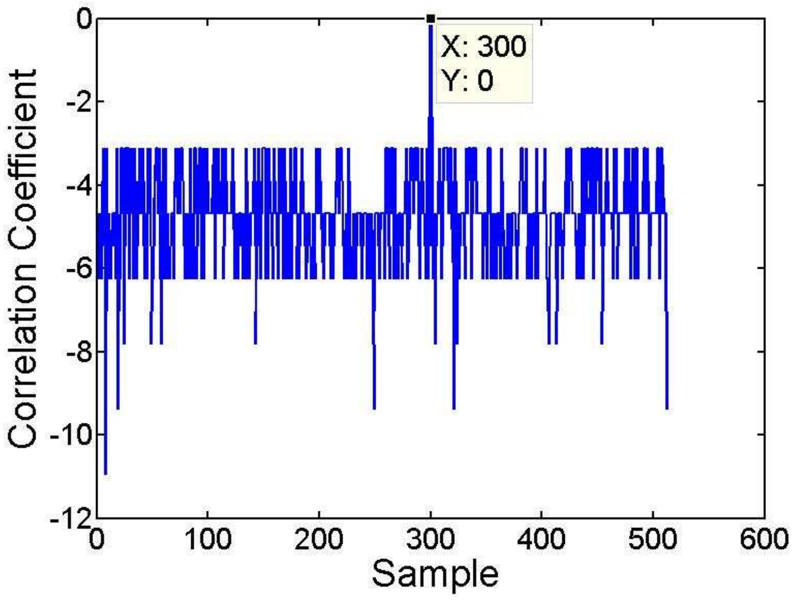


Figure E.11: Cross correlation coefficient value of third combination - using 6 fractional bits.

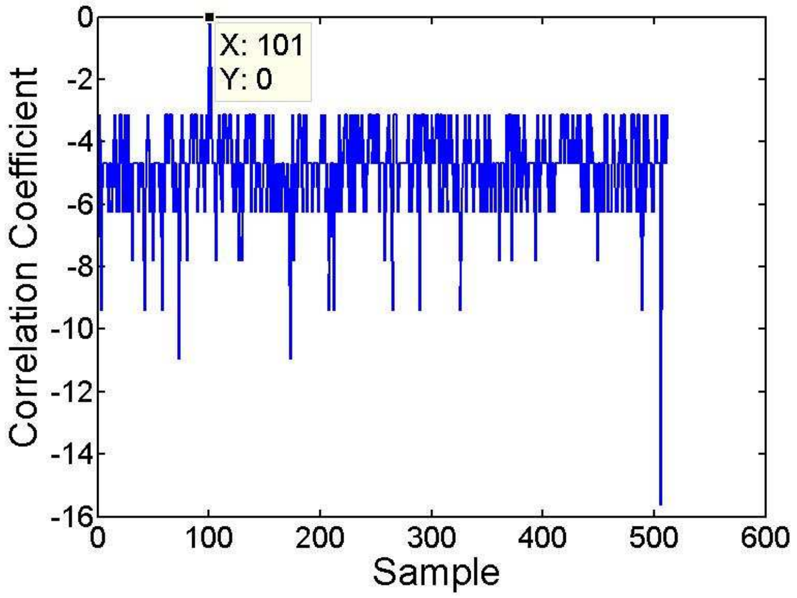


Figure E.12: Cross correlation coefficient value of fourth combination - using 6 fractional bits.

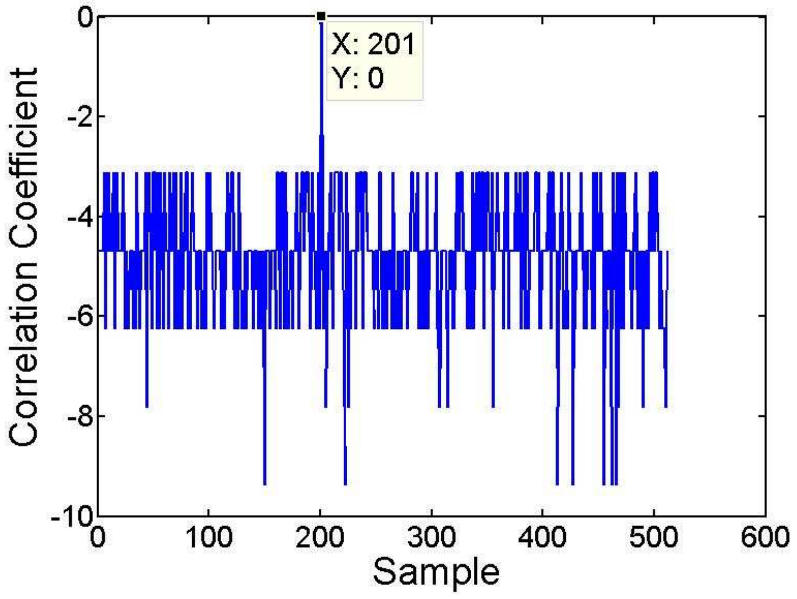


Figure E.13: Cross correlation coefficient value of fifth combination - using 6 fractional bits.

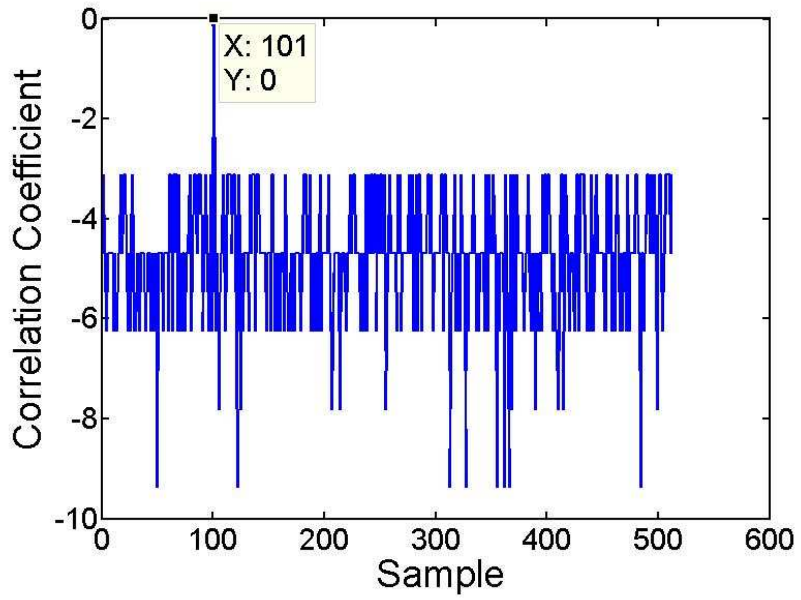


Figure E.14: Cross correlation coefficient value of sixth combination - using 6 fractional bits.

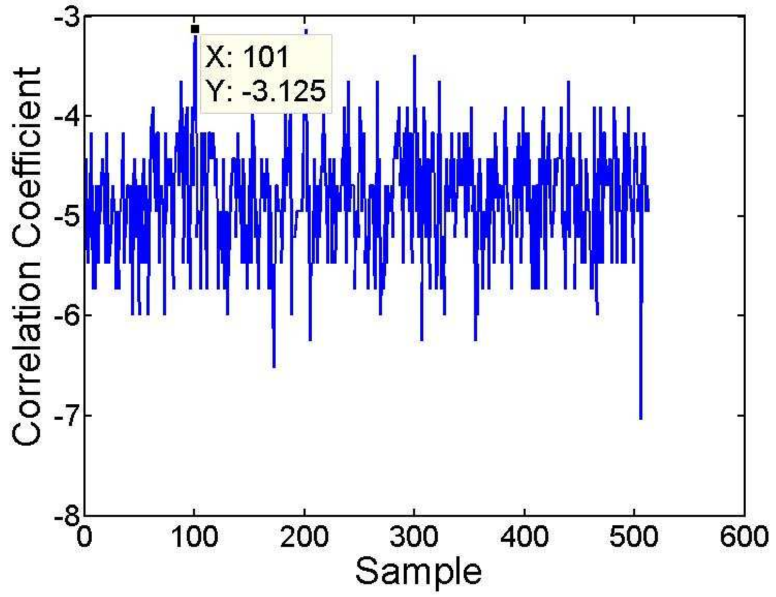


Figure E.15: Log Approximation Based Simple Moving Average of Six combinations - using 6 fractional bits.

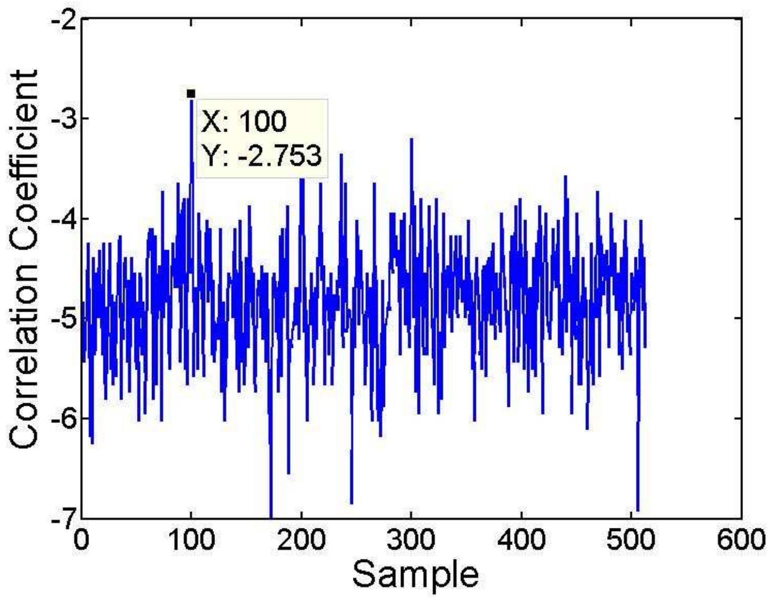


Figure E.16: Log Approximation Based Weighted Moving Average of Six combinations - using 6 fractional bits.

# Appendix F

# VHDL Sample Code

```

entity simtest is
    Port ( clk : in  STD_LOGIC;
--              xrun : in std_logic;

                xxnormout1,xxnormout2,xxnormout3,
xxnormout4,xxnormout5,xxnormout6 :out std_logic_vector(22 downto ...
    0));
--             sdataout : out std_logic;
--              xx : out std_logic_vector(11 downto 0);
--             ja : inout  STD_LOGIC_vector(3 downto 0));
end simtest;

architecture Behavioral of simtest is
signal spiclk, spiclk_ce, newclk1,sdone, xdataout : std_logic  ...
    := '0';
signal sineout : std_logic_vector(9 downto 0):= (others=>'0');
signal  A,B,C,D : std_logic_vector (11 downto 0):= (others=>'0');

signal xsu1, xsd1, xcorr1 : std_logic_vector(35 downto 0):= ...
    (others=>'0');
signal Xnorm1 : std_logic_vector(22 downto 0):= (others=>'0');
signal xzero1, xsign1 : std_logic_vector(0 downto 0):= ...
    (others=>'0');
```

```



signal xsu2, xsd2, xcorr2 : std_logic_vector(35 downto 0):= ...
    (others=>'0');
signal Xnorm2 : std_logic_vector(22 downto 0):= (others=>'0');
signal xzero2, xsign2 : std_logic_vector(0 downto 0):= ...
    (others=>'0');


signal xsu3, xsd3, xcorr3 : std_logic_vector(35 downto 0):= ...
    (others=>'0');
signal Xnorm3 : std_logic_vector(22 downto 0):= (others=>'0');
signal xzero3, xsign3 : std_logic_vector(0 downto 0):= ...
    (others=>'0');

signal xsu4, xsd4, xcorr4 : std_logic_vector(35 downto 0):= ...
    (others=>'0');
signal Xnorm4 : std_logic_vector(22 downto 0):= (others=>'0');
signal xzero4, xsign4 : std_logic_vector(0 downto 0):= ...
    (others=>'0');

signal xsu5, xsd5, xcorr5 : std_logic_vector(35 downto 0):= ...
    (others=>'0');
signal Xnorm5 : std_logic_vector(22 downto 0):= (others=>'0');
signal xzero5, xsign5 : std_logic_vector(0 downto 0):= ...
    (others=>'0');

signal xsu6, xsd6, xcorr6 : std_logic_vector(35 downto 0):= ...
    (others=>'0');
signal Xnorm6 : std_logic_vector(22 downto 0):= (others=>'0');
signal xzero6, xsign6 : std_logic_vector(0 downto 0):= ...
    (others=>'0');



--signal xq,wq : std_logic_vector( 7 downto 0):= (others=>'0');
--signal xavgq : std_logic_vector( 8 downto 0):= (others=>'0');
```

```
signal readdata : std_logic_vector(47 downto 0):= (others=>'0');

signal xusu, xusd,yusu, yusd, ydsu, ydsd,Ausu, Ausd, Adsu, Adsd, ...
    Bdsu, Bdsd :  std_logic_vector(11 downto 0):= (others=>'0');


begin


data: entity work.read_file
     port map (
                    clk =>newclk1,
                    q => readdata);
--------------------------------------------------------------------------------------
--------2mhz clk for pmod ...
    ADC-----------------------------------------------------
--------------------------------------------------------------------------------------

spiclock : entity work.clkdivider -- 2mhz for adc
          port map    (
          clk =>  clk,-- 100 mhz
          dclk => spiclk  );

spiclock_ce : entity work.clkdivider_ce
          port map    (
          clk =>  clk,-- 100 mhz
          dclk => spiclk_ce   );

--------------------------------------------------------------------------------------
--------sampling Frequency ...
    ----------------------------------------------------------
--------------------------------------------------------------------------------------

run_enable : entity work.clkdividersine -- 97.656 khz -- sample ...
    frequency
          port map    (
               clk =>  clk, --100 mhz
```

```
             dclk => newclk1 ); -- 97.656 KHZ

--------------------------------------------------------------------------------------------
---------------upstream downstream ...
    fifo-----------------------------------
--------------------------------------------------------------------------------------------
sig : entity work.signals
    port map ( clk  => clk,
                    sample => newclk1,
                     x => A,
                     y => B,
                     A =>   C,
                     B => D,
                     xusu => xusu,
                     xusd => xusd,
                     yusu => yusu,
                     yusd => yusd,
                     ydsu => ydsu,
                     ydsd => ydsd,
                     Ausu => Ausu,
                     Ausd => Ausd,
                     Adsu => Adsu,
                     Adsd => Adsd,
                     Bdsu => Bdsu,
                     Bdsd => Bdsd );
------------------------------------------------------------------------------------------------
-------cc1--------------------------------------------------
------------------------------------------------------------------------------------------------

cc1ICorr : entity work.ICrr
    port map( clk => clk, -- 100 mhz
                usu => xusu,
                usd => xusd,
                dsu => ydsu,
                dsd => ydsd,
                sample => newclk1, -- 97.656 KHZ
                corr => xcorr1, --corr value
                su =>xsu1,  -- su^2
                sd => xsd1); -- sd^2

cc1Normalization : entity work.operationalblock

         port map( CLK  => clk, --100 mhz;
                        SU   => xsu1,    -- SU^2;
                        SD   => xsd1,   -- SD^2;
                        corr  => xcorr1, -- SU_SD( corr);
                        Norm  => xNorm1,   -- output
                        zero    => xzero1,
                        sign => xsign1);
------------------------------------------------------------------------------
-------cc2----------------------------------------------------
------------------------------------------------------------------------------

cc2ICorr : entity work.ICrr
    port map( clk => clk, -- 100 mhz
                usu => xusu,
                usd => xusd,
                dsu => Adsu,
                dsd => Adsd,
                sample => newclk1, -- 97.656 KHZ
                corr => xcorr2, --corr value
                su =>xsu2,  -- su^2
                sd => xsd2); -- sd^2

cc2Normalization : entity work.operationalblock

         port map( CLK  => clk, --100 mhz;
                        SU   => xsu2,    -- SU^2;
                        SD   => xsd2,   -- SD^2;
                        corr  => xcorr2, -- SU_SD( corr);
                        Norm  => xNorm2,   -- output
                        zero    => xzero2,
                        sign => xsign2);
------------------------------------------------------------------------------
```

```
--------cc3------------------------------------------------------------
-------------------------------------------------------------------------------------------

cc3ICorr : entity work.ICrr
    port map( clk => clk, -- 100 mhz
                usu => xusu,
                usd => xusd,
                dsu => Bdsu,
                dsd => Bdsd,
                sample => newclk1, -- 97.656 KHZ
                corr => xcorr3, --corr value
                su =>xsu3,  -- su^2
                sd => xsd3); -- sd^2

cc3Normalization : entity work.operationalblock

         port map( CLK  => clk, --100 mhz;
                        SU   => xsu3,    -- SU^2;
                        SD   => xsd3,   -- SD^2;
                        corr  => xcorr3, -- SU_SD( corr);
                        Norm  => xNorm3,   -- output
                        zero    => xzero3,
                        sign => xsign3);
-------------------------------------------------------------------------------------------
--------cc4------------------------------------------------------------
-------------------------------------------------------------------------------------------

cc4ICorr : entity work.ICrr
    port map( clk => clk, -- 100 mhz
                usu => yusu,
                usd => yusd,
                dsu => Adsu,
                dsd => Adsd,
                sample => newclk1, -- 97.656 KHZ
                corr => xcorr4, --corr value
                su =>xsu4,  -- su^2
                sd => xsd4); -- sd^2
```

```

cc4Normalization : entity work.operationalblock

         port map( CLK  => clk, --100 mhz;
                        SU   => xsu4,    -- SU^2;
                        SD   => xsd4,   -- SD^2;
                        corr  => xcorr4, -- SU_SD( corr);
                        Norm  => xNorm4,   -- output
                        zero    => xzero4,
                        sign => xsign4);

--------------------------------------------------------------------------------
-------cc5--------------------------------------------------
--------------------------------------------------------------------------------

cc5ICorr : entity work.ICrr
    port map( clk => clk, -- 100 mhz
                usu => yusu,
                usd => yusd,
                dsu => Bdsu,
                dsd => Bdsd,
                sample => newclk1, -- 97.656 KHZ
                corr => xcorr5, --corr value
                su =>xsu5,  -- su^2
                sd => xsd5); -- sd^2

cc5Normalization : entity work.operationalblock

         port map( CLK  => clk, --100 mhz;
                        SU   => xsu5,    -- SU^2;
                        SD   => xsd5,   -- SD^2;
                        corr  => xcorr5, -- SU_SD( corr);
                        Norm  => xNorm5,   -- output
                        zero    => xzero5,
                        sign => xsign5);
--------------------------------------------------------------------------------
-------cc6--------------------------------------------------
```

```
--------------------------------------------------------------------------------

cc6ICorr : entity work.ICrr
    port map( clk => clk, -- 100 mhz
                usu => Ausu,
                usd => Ausd,
                dsu => Bdsu,
                dsd => Bdsd,
                sample => newclk1, -- 97.656 KHZ
                corr => xcorr6, --corr value
                su =>xsu6,  -- su^2
                sd => xsd6); -- sd^2

cc6Normalization : entity work.operationalblock

         port map( CLK  => clk, --100 mhz;
                        SU   => xsu6,    -- SU^2;
                        SD   => xsd6,   -- SD^2;
                        corr  => xcorr6, -- SU_SD( corr);
                        Norm  => xNorm6,   -- output
                        zero    => xzero6,
                        sign => xsign6);



--------------------------------------------------------------------------------
-------Peak detect------------------------------
--------------------------------------------------------------------------------
--peak : entity work.peakdetect
--      port map (
--      clk => clk, -- 100mhz
--      d =>xnorm, --- input
--      q => xq );
----
----
--------------------------------------------------------------------------------
---------avg block------------------------------
```

```
------------------------------------------------------------------------------------------------
--avg : entity work.averageblock
--      port map (
--      clk => newclk1, -- sample
--      d =>xq, --- input
--      q => xavgq );
--
--
------------------------------------------------------------------------------------------------
----------waveform out ...
    ----------------------------------------------------------------------
------------------------------------------------------------------------------------------------
--
--wave : entity work.waveformout
--      port map (
--      clk => clk, -- 100mhz
--      sample => newclk1,
--      d =>xnorm, --- input
--      q => wq);

-----------------------------------------------------------------------------------------------
--------Usb Uart interface For ...
    labview--------------------------------------------
-----------------------------------------------------------------------------------------------

--uartlabview: entity work.main
--          port map (
----              datain => xavgq( 7 downto 0),
--            datain => wq,
--           clk => clk,
--            run => xrun,
--           dataout => xdataout    );


----------------------------------------------------
---- INPUT------------------------------------------

```

```
    A ≤ readdata(47 downto 36);  -- DAC input A
    B ≤ readdata(35 downto 24); -- DAC input B
    C ≤ readdata(23 downto 12);  -- DAC input A
    D ≤ readdata(11 downto 0); -- DAC input B



    xxnormout1 ≤ xnorm1;
    xxnormout2 ≤ xnorm2;
    xxnormout3 ≤ xnorm3;
    xxnormout4 ≤ xnorm4;
    xxnormout5 ≤ xnorm5;
    xxnormout6 ≤ xnorm6;
--   xx ≤ A; -- led on fpga
--  sdataout ≤ xdataout; -- usb uart pin
--  process( spiclk)
--  begin
--  if rising_edge(spiclk) then
--  A ≤ xdoutA; -- DAC input A
--  end if;
--  end process;


end Behavioral;
```