BINARY INTEGER DECIMAL-BASED FLOATING POINT ADDER

By

Ahmed Abdelmordy Mohamed Ayoub

A Thesis Submitted to the
Faculty of Engineeringat Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Electronics and Communications Engineering

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2014

BINARY INTEGER DECIMAL-BASED FLOATING POINT ADDER

By

Ahmed Abdelmordy Mohamed Ayoub

A Thesis Submitted to the
Faculty of Engineeringat Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Electronics and Communications Engineering

Under the Supervision of

Prof. Dr. Amin Mohamed Nassar
Professor
Electronics and Communications Engineering Department
Faculty of Engineering, Cairo University

Dr. Hossam A. H. Fahmy
Associate Professor
Electronics and Communications Engineering Department
Faculty of Engineering, Cairo University

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2014

BINARY INTEGER DECIMAL-BASED FLOATING POINT ADDER

By

Ahmed Abdelmordy Mohamed Ayoub

A Thesis Submitted to the
Faculty of Engineeringat Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Electronics and Communications Engineering

Approved by the Examining Committee:

Prof. Dr. Amin Mohamed Nassar, Thesis Main Advisor

Dr. Hossam A. H. Fahmy, Member

Dr. Ibrahim M. Qamar, Internal Examiner

Prof. Dr. EL-Sayed Mostafa Saad, External Examiner
Professor, Faculty of Engineering, Helwan University

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2014

Acknowledgements

First and foremost, I have to thank God for guiding me to finishing this work.

I offer my sincerest gratitude to my supervisor, Dr. Hossam Fahmy, who has supported me throughout my thesis with his patience and knowledge. I attribute the level of my Masters degree to his encouragement and effort and without him this thesis, too, would not have been completed or written. One simply could not wish for a better or friendlier supervisor. Special thanks also to Dr. Amin Nassar who helps me to complete this work.

I would like to thank SilMinds, SilMinds allowed us to use their owned DFP Adder and DPD to/from BID converter blocks in this research work. I also would like to thank my previous colleague in SilMinds Eng. Tarek El-Deeb for providing help and advice regarding designing DPD to/from BID converters.

I owe a very important debt to my parents. Without their encouragement, I would never reach .. They always encourage and support me.

My deepest appreciation goes to my dear wife for her cooperation, support and patience, she helped me in writing this thesis in LaTeX . I would like to show my greatest appreciation to my wife and my son who sacrificed by their time with me.

Dedication

To the faithful thousands who have been killed in the Egyptian revolution to build newer Egypt.

To Eng. Assem El Gamal - The Design Manager of SilMinds - who is killed in 14/8/2013 to defend his choice and freedom.

Table of Contents

List of Tables

List of Figures

List of Abbreviation

ASIC	Application-Specific Integrated Circuit
BCD	Binary Coded Decimal
BID	Binary Integer Decimal
CSA	Carry Save Adder
DFP	Decimal Floating Point
DPD	Densely Packed Decimal
FPGA	Field Programmable Gate Array
IEEE	Institute of Electrical and Electronics Engineers
Inf	Infinity
LSB	Least Significant Bit
MSB	Most Significant Bit
MUX	Multiplexer
NaN	Not a Number
qNaN	quiet NaN
sNaN	signaling NaN

Abstract

Hardware designs for Decimal Floating point arithmetic have been a hot topic of research in the first decade of the twenty first century.

Decimal Floating Point numbers may use either the decimal encoding referred to as Densely Packed Decimal (DPD) or the binary encoding referred to as Binary Integer Decimal (BID). Most hardware designs so far used the DPD encoding. Hardware units dealing directly with BID encoding were not as efficient.

In this thesis, we propose to use novel converters from BID to DPD and vice-versa to surround the inner core of conventional DPD designs. This conversion from BID followed by the core of a DPD unit and finally a conversion back to BID proved to be a better alternative to direct BID units' implementations. As an example, the decimal floating point adder using our method has a 56.6% reduction in the area, 25% reduction in the power consumption and 4% reduction in Area delay product compared to a published BID adder.

Definitions

- **Biased exponent:** The sum of the exponent and a constant (bias) are chosen to make the biased exponent's range nonnegative.

- **Binary floating-point number:** A floating-point number with radix two.

- **Cohort:** In a given format, the set of floating-point representations with the same numerical value.

- **Decimal floating-point number:** A floating-point number with radix ten.

- **Declet:** An encoding of three decimal digits into ten bits using the densely packed decimal encoding scheme. Of the 1024 possible declets, 1000 canonical declets are produced by computational operations, while 24 noncanonical declets are not produced by computational operations, but are accepted in operands.

- **Exception:** An event that occurs when an operation has no outcome suitable for every reasonable application.

- **Exponent:** The component of a binary floating-point number that normally signifies the integer power to which the radix two is raised in determining the value of the represented number. Occasionally the exponent is called the signed or unbiased exponent.

- **Floating-point number:** A bit-string encoding characterized by three components: a sign, a signed exponent, and a significand. All floating-point numbers, including zeros and infinities, are signed.

- **Format:** A set of representations of numerical values and symbols, perhaps accompanied by an encoding.

- **NaN:** Not a Number, a symbolic entity encoded in floating-point format. There are two types of NaNs , quiet and signaling. quiet NaNs propagate through almost every arithmetic operations without signaling exceptions, while signaling NaNs signal the invalid operation exception whenever they appear as operands.

- **Normal number:** For a particular format, a finite non-zero floating-point number with magnitude greater than or equal to a minimum $b^{e_{min}}$ value, where b is the radix. Normal numbers can use the full precision available in a format. In IEEE Std 754-2008, zero is neither normal nor subnormal.

- **Precision:** The maximum number p of significant digits that can be represented in a format, or the number of digits to that a result is rounded.

- **Signal:** When an operation has no outcome suitable for every reasonable application, that operation might signal one or more exceptions by invoking the default or user specified alternate handling. Note that exception and signal are defined in diverse ways in different programming environments.

- **Significand:** A component of an unencoded binary or decimal floating-point number containing its significant digits. The significand may be thought of as an integer, a fraction, or some other fixed-point form, by choosing an appropriate bias. The component of a binary floating-point number that consists of an explicit or implicit leading bit to the left of its implied binary point and a fraction field to the right.

- **Subnormal number:** In a particular format, a non-zero floating-point number with magnitude less than the magnitude of that formats smallest normal number. A subnormal number does not use the full precision available to normal numbers of the same format.

- **Trailing significand field:** A component of an encoded binary or decimal floating-point format containing all the significand digits except the leading digit. In these formats, the biased exponent or combination field encodes or implies the leading significand digit.

- **Quiet operation:** Any of the operations specified by IEEE Std 754-2008 that never generate an exception.

Chapter 1: Introduction

1.1 Background

Although most people use decimal arithmetic when performing manual calculations, computers typically only support binary arithmetic in hardware. This is primarily due to there being only two logic values, zero and one, that are represented in modern computers.

While it is possible to use these two logic values to represent decimal numbers, doing so is wasteful in terms of storage space and is also less efficient. For example, in binary, four bits can represent sixteen values; while in binary coded decimal (BCD), four bits only represent ten values. Since most computer systems do not provide hardware support for decimal arithmetic, numbers are typically input in decimal, converted from decimal to binary, processed using binary arithmetic, and then converted back to decimal for output.

In spite of the current dominance of hardware support for binary arithmetic, there are several motivations that encourage the provision of support for decimal arithmetic. First, applications that deal with financial and other real-world data often have errors introduced, since many common decimal numbers cannot be represented exactly in binary. For example, the decimal number 0.1 is a repeating fraction when represented in binary. Second, people typically think about computations in decimal, even when using computers that operate only on binary representations, and therefore may experience what is perceived as incorrect behavior when processing decimal values. Third, converting between binary and decimal floating-point numbers is computationally intensive and may take thousands of cycles on modern processors.[40]

Because of the growth interest in the decimal computations, IEEE extended the previous standard (IEEE Std 754-1985) [2] , and added decimal arithmetics and formats in the current version (IEEE Std 754-2008) [4] of IEEE Standard for Floating-Point Arithmetic (IEEE 754).

Decimal Floating-Point (DFP) computations are critical for many financial and commercial applications. With trends towards globalization, many laws and standards require decimal calculations. For example, the European Union requires currency conversion to and from the Euro to be calculated to six decimal places. One study estimates that a large telephone billing system can accumulate errors of up to 5 million US Dollar per year, if using binary floating-point arithmetic, rather than decimal. Both hardware and software solutions for DFP arithmetic are being developed to remedy these problems [48].

Also, another important question is why do we need to replace the existing software conversion from decimal to BCD than back to decimal into hardware. An interesting study [21] shows that application can realize performance improvements ranging from about 10% (for applications whose respective DFP routines consumes10% of the execution time) to nearly 1000% (for applications whose respective DFP routines consumes 90% of the execution time).

Due to the rapid growth in financial, commercial, and Internet-based applications, there is an increasing desire to allow computers to operate on both binary and decimal floating-point numbers. Consequently, specifications for decimal floating- point arithmetic were added to the IEEE-754 Standard for Floating-Point Arithmetic which was published in 1985. In this thesis, we present the design and implementation of a decimal floating-point adder/subtractor that is compliant with the current version (IEEE Std 754-2008) [4] of IEEE Standard for Floating-Point Arithmetic (IEEE 754). The adder supports operations on 64-bit (16- digit) and 128-bit (34-digit) decimal floating-point operands. We provide two different architectures for the adder/subtractor. Synthesis results indicating the area usage and the clock frequency with Stratix XI Altera FPGA will be introduced in chapter 4.

1.2 Problem description

Most computers support binary floating point arithmetic, the main advantage of the current decimal floating point is the accuracy. The most challenging issues are the performance and the area of the decimal floating point units. Binary floating-point numbers can only approximate common decimal numbers. The value 0.1, for example, would need an infinitely recurring binary fraction. In contrast, a decimal number system can represent 0.1 exactly, as one tenth (that is, 10^{-1}). Consequently, binary floating-point cannot be used for financial calculations or indeed for any calculations where the results achieved are required to match those which might be calculated by hand. Table1.1 demonstrates dividing the number nine repeatedly divided by ten.

Table 1.1: Dividing the number nine repeatedly by ten

Binary	Decimal
0.9	0.9
0.089999996	0.09
0.0090	0.009
9.0E-4	0.0009
9.0E-5	0.0000
9.0E-6	0.000009
9.0000003E-7	9E-7
9.0E-8	9E-8
9.0E-9	9E-9
8.9999996E-10	9E-10

Here, the right hand column shows the results delivered by decimal floating-point arithmetic (such as the BigDecimal class for Java [33] or the decNumber C package [18]), and the left hand column shows the results obtained by using the Java float data type. The results from using the double data type are similar to the latter (with more repeated 9s or 0s).

For these reasons, most commercial data are held in a decimal form. Calculations on decimal data are carried out using decimal arithmetic, almost invariably operating on

numbers held as an integer and a scaling power of ten. The IBM z900 computer range was the only widely-used computer architecture with built-in decimal arithmetic instructions. However, those early instructions work with decimal integers only, which then require manually applied scaling. This is error- prone, difficult to use, and hard to maintain, and requires unnecessarily large precisions when both large and small values are used in a calculation. Both problems (the artifacts of binary floating-point, and the limitations of decimal fixed-point) can be solved by using a decimal floating-point arithmetic. This is now available in both hardware and software. and is standardized in the IEEE Standard for Floating-Point Arithmetic (IEEE 754). But the other problem in decimal floating point units is the complexity comparing to the binary floating point units, this complexity can be translated to larger area and lower performance, in our proposed design, we implemented new design of the decimal floating point adder which has some better results for the area and performance than the published work [43] related to hardware implementation of decimal floating point adder.

1.3 The Decimal Floating Point Formats

IEEE Std 754-2008 [4] standardizes the encodings of binary and decimal floating point data, albeit with two different alternative encodings.

For decimal floating point data, two different encodings are defined:

- Binary Integer Decimal (BID) encodes the significand as a large binary integer between 0 and $(10^p - 1)$ where P is the coefficient size or the number of the decimal digits. The value of Decimal Floating Point number is :

  $(-1)^S \times 10^{E-bias} \times C$, where S is the sign bit, E is a biased exponent, $bias$ is a constant value that makes E non-negative. $C = (d_{p-1}d_{p-2}\ldots d_0)$ is the significand where $d \in 0;1;2;3;4;5;6;7;8;9$, and p is the precision.

  This is expected to be more convenient for software implementations using a binary ALU.

- Densely Packed Decimal (DPD) encodes decimal digits more directly. This is expected to be more convenient for hardware implementations. [52]

The general decimal floating point format [4] is shown in figure 1.1, this general format includes BID and DPD:

where S is the sign, G-field is a combination field that contains the exponent, the most significant digit of the significand, and the encoding classification, T hold most of the operand significand (Trailing significand field) and it is decoded to $P-1$ BCD digits, t is the number of bits in the significand field and J is the number of declets(Each declet contains three digits) in the significand fields.

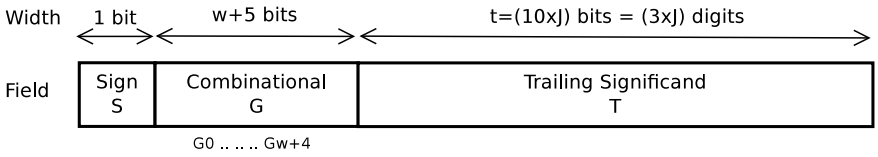


Figure 1.1: Decimal Floating-Point Interchange Format

Binary integer decimal encoding is shown with another divisions for more clarification in figure 1.2 and 1.3:

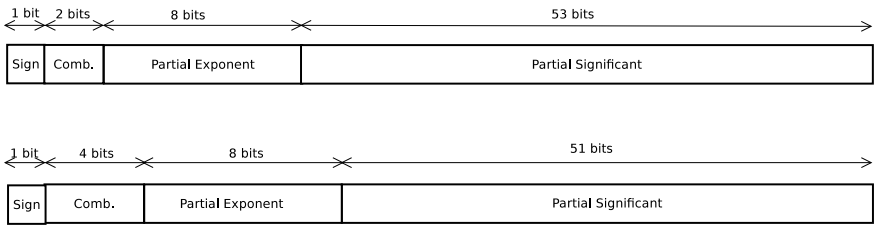


Figure 1.2: Binary Integer Decimal Encoding - 64 bits

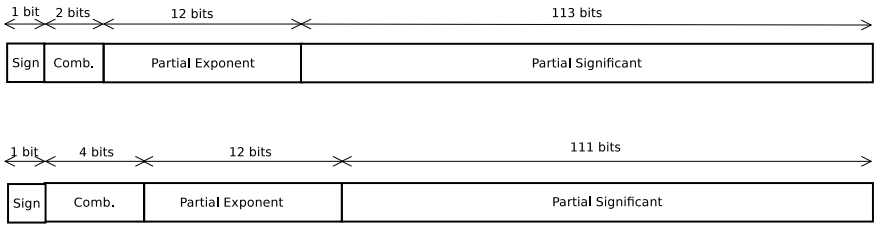


Figure 1.3: Binary Integer Decimal encoding - 128 bits

An example, the Decimal64 significand can be up to $(10^{16}-1) = 9\,999\,999\,999\,999\,999$ = 23 86F2 6FC0 FFFF (in Hex 54bits) = 10 0011 1000 0100 1111 0010 0110 1111 1100 0000 1111 1111 1111 1111 (in binary 54 bits) while the encoding can represent larger significands, they are illegal (non-canonical) and the standard requires implementations to treat them as 0, if encountered on input. The most significant four bits of the significand has maximum value of "1000".

The exponent ranges were chosen so that the most significant two bits for the stored exponent E has value of "10". The stored exponent values are biased by $bias$ – which is 398 – to make sure that all available values are positive (unsigned numbers).

For decimal64, Largest possible value is $9.999\,999\,999\,999\,999 \times 10^{384}$, it can be written as $9\,999\,999\,999\,999\,999 \times 10^{369}$ to write integer significand number. The max exponent which can be stored is 369. The stored biased exponent equals $369 + 398 = 767(integer) = 2FF(Hex)$, the most significant two bits for the maximum stored biased exponent value are "10", so the most significant two bits has possible values of ("00", "01" and "10").

For decimal128, Largest possible value is $9.999\ 999\ 999\ 999\ 999\ 999\ 999\ 999\ 999\ 999 \times 10^{6\ 144}$, it can be written as $9\ 999\ 999\ 999\ 999\ 999\ 999\ 999\ 999\ 999\ 999 \times 10^{6\ 111}$ to write integer significand number. the max exponent which can be stored is 6 111. The stored biased exponent equals $6\ 111 + 6\ 176 = 12\ 287(integer) = 2FFF(Hex)$, the most significant two bits for the maximum stored biased exponent value are "10", so the most significant two bits has possible values of ("00", "01" and "10").

The encoding varies depending on whether the most significant 4 bits of the significand are in the range 0 to 7 - decimal (0000 to 0111 - binary), or higher (1000 or 1001 - binary).

If the two bits after the sign are "00","01'" or "10", then the exponent field (from G0 to G9) consists of the 10 bits following the sign bit (the two mentioned bits plus eight bits of "the partial exponent") and the significand is the remaining 53 bits, with an implicit leading 0 bit, shown here in Figure 1.4:

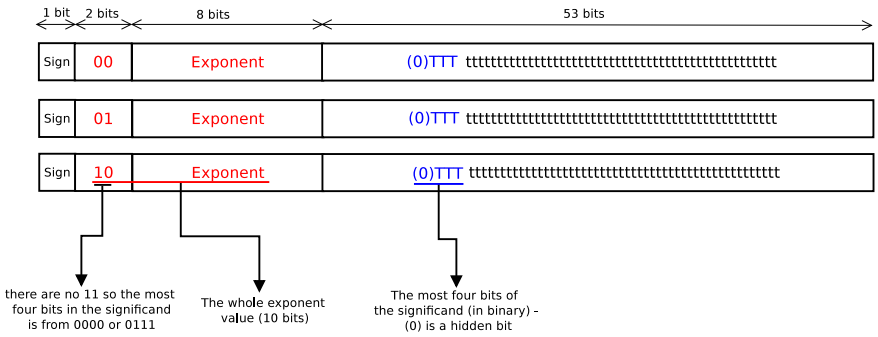


Figure 1.4: Binary Integer Decimal Encoding format for the most significant digit from 0 – 7, 64bits

This includes subnormal numbers where the most significant digit is 0.

If the 4 bits after the sign bit are "1100", "1101", or "1110", then the 10-bit exponent field (biased exponent) is shifted 2 bits to the right (after both the sign bit and the "11" bits thereafter) so it will be from G2 to G11, and the represented significand (partial significand) is in the remaining 51 bits. In this case there is an implicit (that is not stored) leading 3-bit sequence "100" in the true significand:

The "11" 2–bit sequence after the sign bit indicates that there is an implicit "100" 3-bit prefix to the trailing significand.

Densely Packed Decimal fields are shown in figure 1.6 and 1.7.

The most significant two bits of the exponent are limited to the range of 0 – 2, and the most significant 4 bits of the significand are limited to the range of 0 – 9.

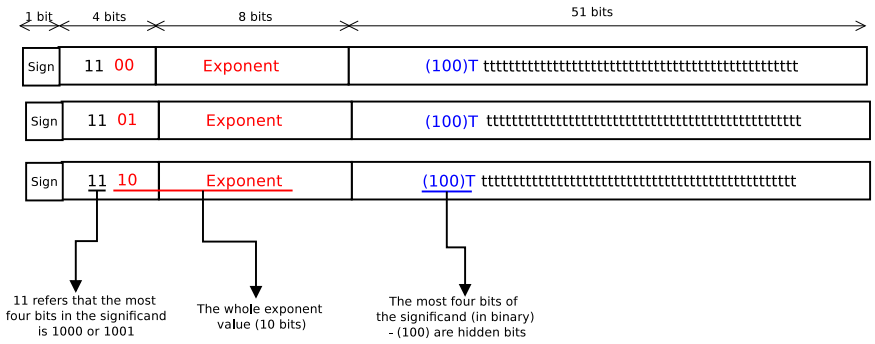


Figure 1.5: Binary Integer Decimal Encoding format for the most significant digit from 8 – 9, 64 bits

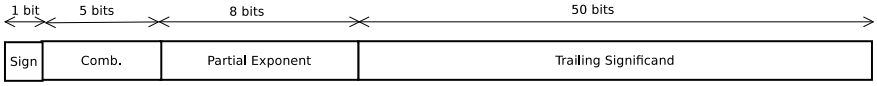


Figure 1.6: Densely Packed Decimal encoding, 64 bits

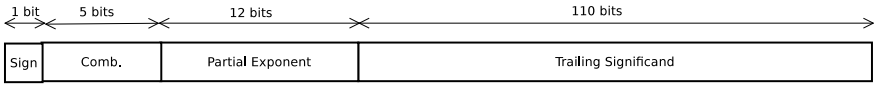


Figure 1.7: Densely Packed Decimal encoding, 128 bits

For a decimal number :

$$d_{15}\,,d_{14}\,d_{13}\,d_{12}\,,d_{11}\,d_{10}\,d_9,d_8d_7d_6,d_5d_4d_3,d_2d_1d_0$$

If the most significant digit d_{15} is between 0 to 7, The encoded value is represented as follows :

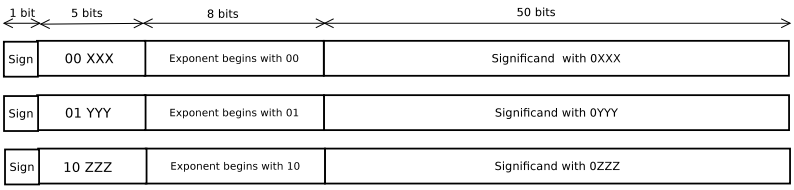


Figure 1.8: DPD64 Encoding format for the most significant digit from 0 – 7

For a decimal number :

$$d_{15}\,,d_{14}\,d_{13}\,d_{12}\,,d_{11}\,d_{10}\,d_9,d_8d_7d_6,d_5d_4d_3,d_2d_1d_0$$

If the most significant digit d_{15} is binary 1000 or 1001 (decimal 8 or 9), The number is represented as follows :

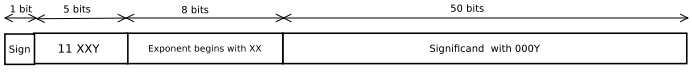


Figure 1.9: DPD64 Encoding format for the most significant digit from 8 – 9

1.3.1 Numerical examples on Binary Integer Decimal Encoding

1.3.1.1 Example 1

If we have 1.23×10^{-2} float number, we need to write it as the form of $(-1)^S \times 10^{E-bias} \times C$. The most significant 4 bits for C are "0000" which is between decimal 0 – 7 so the first BID encoding format will be used.

$N = 1.23 \times 10^{-2}$
$N = (-1)^0 \times 123 \times 10^{-4}$

$S = 0$	$The\ Significand$ = 0 000 000 000 000 123 (decimal)
	= 00 0000 0000 007$B(Hexadecimal - 54bits)$
Exponent = $-4 = E - 398$	$Trailing\ Significand$
	= 000 000 000 000 123 (in BCD)
$E = 394(decimal) = 1\ 8A(Hexadecimal)$	The most significant 4 bits = **0000**

0 01 1000 1010 **000** 00 0000 0000 0000 0000 0000 0000 0000 0000 0000 0111 1011
BID Encoding format = 3140 0000 0000 007B (Hexadecimal)

1.3.1.2 Example 2

If we have $-9.830\ 000\ 000\ 000\ 001\ \times 10^{20}$ float number, we need to write it as the form of $(-1)^S \times 10^{E-bias} \times C$.

The most significant 4 bits for the significand are "1000" which is between decimal 8 – 9 so the second BID encoding format will be used.

$N = -9.830000000000001 \times 10^{20}$
$N = (-1)^1 \times 9\ 830\ 000\ 000\ 000\ 001 \times 10^5$

$S = 1$	$TheSignificand = 9\ 830\ 000\ 000\ 000\ 001(decimal)$
	$= 22\ EC55\ 3A24\ 6001(Hexadecimal - - - 54bits)$
Exponent = $5 = E - 398$	$Trailing\ Significand$ = 830 000 000 000 001 (in BCD)
$E = 403(decimal)1\ 93(Hexadecimal)$	The most significant 4 bits = **1000**

1 11 01 1001 0011 **0** 10 1110 1100 0101 0101 0011 1010 0010 0100 0110 0000 0000 0001
BID Encoding format = ECA2 EC55 3A24 6001 (Hexadecimal)

1.3.1.3 Example 3

If we have -0.005×10^{105} float number, we need to write it as the form of $(-1)^S \times 10^{E-bias} \times C$.

The most significant 4 bits for the significand are "0000" which is between decimal 0 – 7 so the first BID encoding format will be used.

$N = -0.005 \times 10^{105}$
$N = (-1)^1 \times 5 \times 10^{102}$

S = 1	$TheSignificand$ = 0 000 000 000 000 005 (decimal) = 00 0000 0000 0005($Hexadecimal---54bits$)
Exponent = 102 = $E-398$	Trailing Significand = 000 000 000 000 005 (in BCD)
E = 500($decimal$) = 1 $F4$($Hexadecimal$)	The most significant 4 bits = **0000**

1 01 1111 0100 **000** 00 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0101

BID Encoding format = BE80 0000 0000 0005 (Hexadecimal)

1.3.1.4 Example 4

In this example we have -0.0050×10^{105} float number. -0.0050×10^{105} has trailing zero which can not be ignored, this number is not the same of -0.005×10^{105} in example 3 so we need a different representation for this number. we need to write it as the form of $(-1)^S\times10^{E-bias}\times C$.

The most significant 4 bits for the significand are "0000" which is between decimal 0 – 7 so the first BID encoding format will be used.

$N=-0.0050\times10^{105}$
$N=(-1)^1\times50\times10^{101}$

S = 1	$TheSignificand$ = 0 000 000 000 000 050 (decimal) = 00 0000 0000 0032($Hexadecimal---54bits$)
Exponent = 101 = $E-398$	$TrailingSignificand$ = 000 000 000 000 050 (in BCD)
E = 499($decimal$) = 1 $F3$($Hexadecimal$)	The most significant 4 bits = **0000**

1 01 1111 0011 **000** 00 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0011 0010

BID Encoding format = BE60 0000 0000 0032 (Hexadecimal)

1.3.1.5 Example 5

In this example, we have $9\,999\,999\,999\,999\,999\times10^{250}$ float number, we need to write it as the form of $(-1)^S\times10^{E-bias}\times C$.

C has the maximum possible decimal number in BID64 encoding, The most significant 4 bits in this case are "1000" which is between decimal 8 – 9 so the second BID encoding format will be used.

$N=9\,999\,999\,999\,999\,999\times10^{250}$
$N=(-1)^0\times9\,999\,999\,999\,999\,999\times10^{250}$

S = 0	The Significand = 9 999 999 999 999 999 (decimal) = 23 86$f2$ 6fc0 $FFFF$ (Hexadecimal - 54 bits)
Exponent = 250 = $E-398$	Trailing Significand = 999 999 999 999 999 (in BCD)
E = 648($decimal$) = 2 88($Hexadecimal$)	The most significant 4 bits = **1000**

0 11 10 1000 1000 **0** 11 1000 0110 1111 0010 0110 1111 1100 0000 1111 1111 1111 1111

BID Encoding format = 7443 86F2 6FC0 FFFF (Hexadecimal)

1.3.1.6 Example 6

If we have $-0.000\,000\,000\,000\,000\,000\,000\,001$ float number, we need to write it as the form of $(-1)^S \times 10^{E-bias} \times C$. This number has more than 16 precision digits so BID128 encoding format will represent it.

The most significant 4 bits for the significand are "0000" which is between decimal 0 – 7 so the second BID encoding format will be used.

$N = -0.000\,000\,000\,000\,000\,000\,000\,001$
$N = (-1)^1 \times 1 \times 10^{-24}$

$S = 1$	*The Significand* = 0 000 000 000 000 000 000 000 000 000 001 (decimal) = 0 0000 0000 0000 0000 0000 0000 0001 (Hex. 114 bits)
Exponent = $-24 = E - 6\,167$	*Trailing Significand* = 000 000 000 000 000 000 000 000 000 001 (in BCD)
$E = 6143(decimal) = 17FF(Hexadecimal)$	The most significant 4 bits = **0000**

1 01 0111 1111 1111 **000** 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001

BID Encoding format = AFFE 0000 0000 0000 0000 0000 0000 0001 (Hexadecimal)

1.3.2 Numerical examples on Densely Packed Decimal Encoding

In this section, we put the same float numbers we mentioned in BID numerical examples in the previous subsection 1.3.1 :

1.3.2.1 Example 7

If we have 1.23×10^{-2} float number, we need to write it as the form of $(-1)^S \times 10^{E-bias} \times C$.

The the most significant digit for the significand is "0000" which is between decimal 0 – 7 so the first DPD encoding format as shown in figure 1.8 will be used.

$N = 1.23 \times 10^{-2}$
$N = (-1)^0 \times 123 \times 10^{-4}$

$S = 0$	C = 0 000 000 000 000 123 (decimal)
Exponent = $-4 = E - 398$	Trailing Significand = 000 000 000 000 123 (decimal)
E = 394 (decimal) = 1 8A(*Hexadecimal*)	The most significant digit = **0000**

0 01 **000** 1000 1010 0000000000 0000000000 0000000000 0000000000 0010100011

DPD Encoding format = 3140 0000 0000 007B (Hexadecimal)

1.3.2.2 Example 8

If we have $-9.830\ 000\ 000\ 000\ 001 \times 10^{20}$ float number, we need to write it as the form of $(-1)^S \times 10^{E-bias} \times C$.

The most significant digit for the significand is "1001" which is between decimal 8 – 9 so the second DPD encoding format as shown in figure 1.9 will be used.

$N = -9.830\ 000\ 000\ 000\ 001 \times 10^{20}$
$N = (-1)^1 \times 9\ 830\ 000\ 000\ 000\ 001 \times 10^5$

S = 1	C = 9 830 000 000 000 001 (decimal)
Exponent = 5 = E – 398	Trailing Significand = 830 000 000 000 001 (decimal)
E = 403 (decimal)	The most significant digit = **1001**
= 1 93(*Hexadecimal*)	

1 11011 1001 0011 0000111100 0000000000 0000000000 0000000000 0000000001
DPD Encoding format = EE4C 3C00 0000 0001 (Hexadecimal)

1.3.2.3 Example 9

If we have -0.005×10^{105} float number, we need to write it as the form of $(-1)^S \times 10^{E-bias} \times C$.

The most digit for the significand is "0000" which is between decimal 0 – 7 so the first DPD encoding format will be used.

$N = -0.005 \times 10^{105}$
$N = (-1)^1 \times 5 \times 10^{102}$

S = 1	C = 0 000 000 000 000 005 (decimal)
Exponent = 102 = E – 398	Trailing Significand = 000 000 000 000 005 (decimal)
E = 500 (decimal)	The most significant digit = **0000**
= 1 F4 (Hexadecimal)	

1 01000 1111 0100 0000000000 0000000000 0000000000 0000000000 0000000101
DPD Encoding format = A3D0 0000 0000 0005 (Hexadecimal)

1.3.2.4 Example 10

Here is the same number -0.0050×10^{105} float number in example 9 with different representation.

This number has a trailing zero digit, that should be preserved. This is to keep the significance value. The exponent should be "1F3" (in Hex). The most digit for the significand is "0000" which is between decimal 0 – 7 so the first DPD encoding format as shown in figure 1.8 will be used.

$N = -0.0050 \times 10^{105}$
$N = (-1)^1 \times 50 \times 10^{101}$

S = 1	C = 0 000 000 000 000 050 (decimal)
Exponent = 101 = E – 398	Trailing Significand = 000 000 000 000 050 (decimal)
E = 499 (decimal)	The most significant digit = **0000**
= 1 F3 (Hexadecimal)	

1 01**000** 1111 0011 0000000000 0000000000 0000000000 0000000000 0001010000

DPD Encoding format = A3CC 0000 0000 0050 (Hexadecimal)

1.3.2.5 Example 11

In this example, we have $9\,999\,999\,999\,999\,999 \times 10^{250}$ float number, we need to write it as the form of $(-1)^S \times 10^{E-bias} \times C$. C has the maximum possible decimal number in BID64 encoding .

The most digit in this case is "1000" which is between decimal 8 – 9 so the second DPD encoding format as shown in figure 1.9 will be used.

$N = 9\,999\,999\,999\,999\,999 \times 10^{250}$

$N = (-1)^0 \times 9\,999\,999\,999\,999\,999 \times 10^{250}$

$S = 0$	$C = 9\,999\,999\,999\,999\,999$ (decimal)
Exponent = 250 = $E - 398$	Trailing Significand = 999 999 999 999 999 (decimal)
$E = 648$ (decimal)	The most significant digit = **1001**
$= 2\ 88$ (Hexadecimal)	

0 11101 1000 1000 0011111111 0011111111 0011111111 0011111111 0011111111

DPD Encoding format = 7620 FF3F CFF3 FCFF(Hexadecimal)

1.3.2.6 Example 12

If we have $-0.000\,000\,000\,000\,000\,000\,000\,001$ float number, we need to write it as the form of $(-1)^S \times 10^{E-bias} \times C$. This number has more than 16 precision digits so BID128 encoding format will represent it.

The most significant digit for the significand is "0000" which is between decimal 0 – 7 so the first DPD128 encoding format will be used.

$N = -0.000\,000\,000\,000\,000\,000\,000\,001$

$N = (-1)^1 \times 1 \times 10^{-24}$

$S = 1$	$C = 0\,000\,000\,000\,000\,000\,000\,000\,000\,001$ (decimal)
Exponent = $-24 = E - 6\,167$	Trailing Significand =
	$000\,000\,000\,000\,000\,000\,000\,000\,001$ (decimal)
$E = 6143$ (decimal)	The most significant digit = **0000**
$= 17\ FF$ (Hexadecimal)	

1 01 **000** 0111 1111 1111 0000000000 0000000000 0000000000 0000000000 0000000000 0000000000 0000000000 0000000000 0000000000 0000000000 0000000001

BID Encoding format = A1FF C000 0000 0000 0000 0000 0000 0001 (Hexadecimal)

1.4 Related Work

1.4.1 The Decimal Floating Point Adders

The decimal-encoded formats and arithmetic described in the new standard now have many implementations in hardware and software including:

- Fujitsus decimal Densely Packed Decimal and NUMBER support instructions in the SPARC64 X processor announced at Hot Chips 24, August 2012. [5]
- SilMinds Decimal Floating Point Arithmetic hardware IP Cores Family. Two hardware implementations are introduced for decimal floating point adder that is compliant with the IEEE 754-2008 Standard; one for High-Speed applications and the other for Low Power/Area ones. [22]
- The paper: Decimal floating-point support on the IBM System z10 processor by Eric Schwarz, John Kapernick, and Mike Cowlishaw [50] describes decimal floating-point hardware in, and supporting software for, the new IBM System z10 mainframe.
- The hardware decimal floating-point unit in the IBM Power6 processor, the firmware (with assists) in the IBM System z9 (mainframe) processor, and the hardware decimal floating-point unit in the IBM System z10 mainframe which is the first mainframe with hardware support for the DFP format in the IEEE 754-2008 floating-point standard. [38]
- Benchmark suite of financial Decimal Floating-Point (DFP) applications. The

  benchmark suite includes a banking benchmark, a Euro conversion benchmark, a risk management benchmark, a tax preparation benchmark, and a telephone billing benchmark. The benchmark suite is being made publicly available. [48]
- IBM XL C/C++ for AIX, Linux and z/OS, DB2 for z/OS, Linux, UNIX, and Windows, and Enterprise PL/I for z/OS; IBM is also adding support to many other software products including z/VM V5.2, System i/OS, the dbx debugger, and Debug Tool Version 8.1.
- SAP Net Weaver 7.1, which includes the new DECFLOAT data dtype in ABAP, with support for hardware decimal floating-point on Power6.
- GCC 4.2 was released in July 2007; this is the first GCC release with support for the proposed ISO C extensions for decimal floating point. [3]

1.4.2 Converters

The Binary to/from BCD converters are important components in our proposed design, these converters are used in the BID to/from BCD converters. In this section, prior art for the other algorithms are presented.

1.4.2.1 Binary to BCD Conversion

We have not encountered any hardware implementation for Binary to BCD conversion for large number of binary bits (more than 50 bits).

1.4.2.1.1 Algorithms for large binary numbers but not efficient: Couleurs algorithm [14] is the first binary to BCD and BCD to binary conversion algorithm. It is known as BIDEC algorithm. This conversion algorithm requires two operations per binary bit. Benedek [8] used Couleur's algorithm to extend the conversion for large binary numbers. He developed a geometrical expansion scheme using static decoders. The other work in [8] is merging groups of decoders with identical I/O pattern to reduce hardware complexity. He considered that this structure is valid till 30 binary input bits.

The algorithm description There are more than one method to convert from binary to BCD form. BIDEC method [14] is one of these methods which is classified as a static conversion. In this design we present a static binary to BCD conversion for large number of binary bits, we designed binary to BCD converters for 54 or 114 binary bit.

The static binary/BCD conversion technique [8] consists of looking at the three most significant bits of the binary number. If the value of the three bits is greater than/or equal to five, add binary three to the number and shift the result one bit to the left. If the three bits value is less than five, shift to the left without adding three. Take the next bit from the right and repeat the operation until reaching the least significant bit which will be carried over without decoding.

We can either use a combinational logic to implement this function or we can use a look-up table elements that will be cascaded to implement this function. The content of the look-up table is shown in Table 1.2.

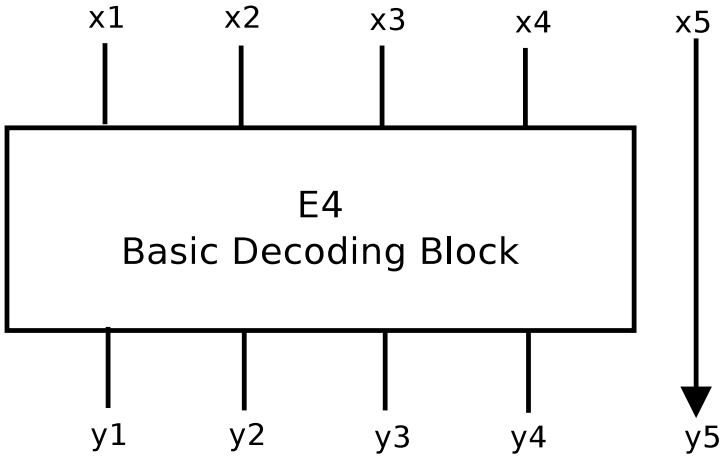


Figure 1.10: E4 Block Entity

x1	x2	x3	x4	y1	y2	y3	y4
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	1
0	1	0	0	0	1	0	0
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

Table 1.2: Look-Up Table Content for E4

Hex	Te	Units	E
Start			1110
Shift 1		1	110
Shift 2		11	10
Shift 3		111	0
Add 3		1010	0
Shift 4	1	0100	
BCD	**1**	**4**	

Figure 1.11 shows the interconnection of these basic (E4) building blocks to form a 11 binary bits to BCD decoder tree. The number of decoder elements increases rapidly with the bits to be converted. Number of stages follows this function:

$$Number\ of\ E4\ decoding\ tree\ stages\ = Number\ of\ Input\ bits\ - 3$$

An example of shift and add-3 algorithm is shown above, The binary number "1110" can be converted to its value in BCD "14" through these steps:

1. Check the 4 bits number in the Hundreds, Tens, and Units. If they are greater or equal to 5, then add 3 to it.
2. Shift the binary number left one bit.
3. Repeat from the top. (if you are converting a 8 bit binary number, you do it 8 times as the number of bits equals 8; if 12 bit then 12 times.

An example of binary to BCD conversion using E4 decoding tree is shown in figure 1.12. The 11 binary input bits are 10011110101, the corresponding decimal (BCD) number is 1269. Using table 1.2, we can get this BCD number.

Developing the E6 level of Interconnect In [8], they developed a higher level decoder by merging three units of Figure 1.11 into one (dotted line), the new decoding unit of (E6) has six input/six outputs. The truth table for this unit is shown in Table 1.3, the three least significant output bits have only 5 out of 8 combinations from 0 to 4, these three bits will be connected to the three most significant input bits of another decoding unit. The three least input bits have all possible values (8 numbers) so the total number of words stored will be 5 x 8 = 40 words.

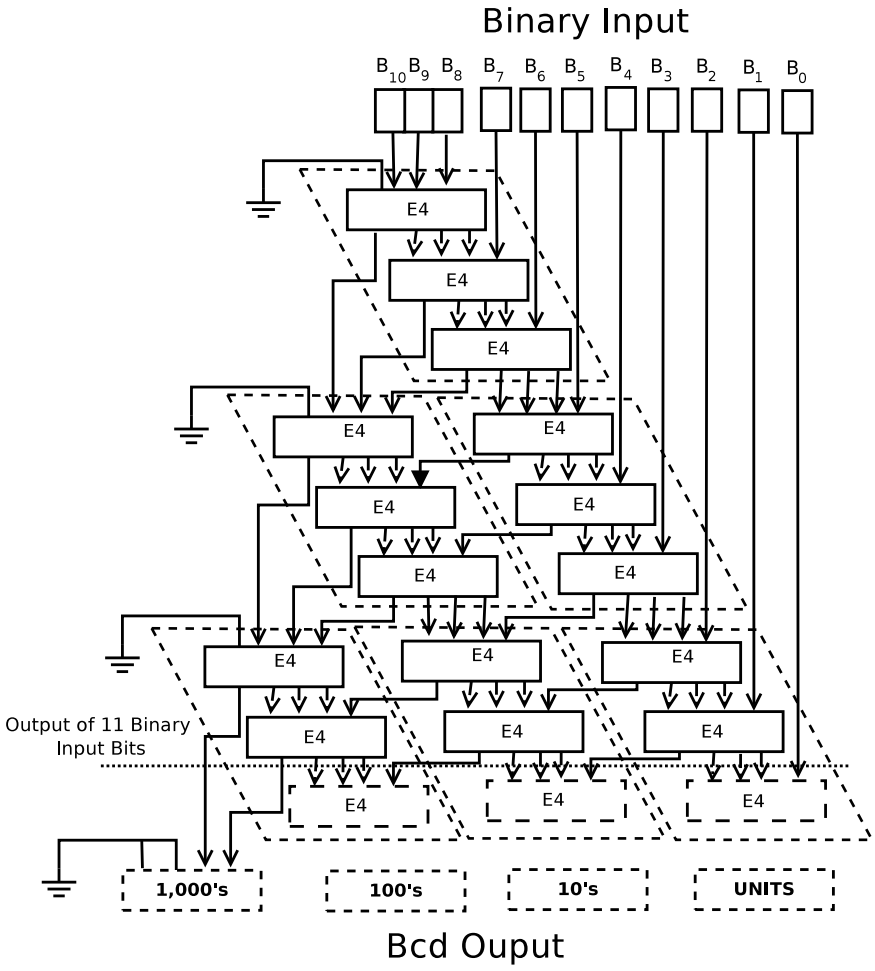


Figure 1.11: 11 Binary bits to BCD decoding tree with E4 decoders, the dotted outlines represent E6 decoders

The advantage of this map is that the decoding tree can be extended to any number of bits by topological similarity and can determine the number of decoder units required for a given number of binary bits.

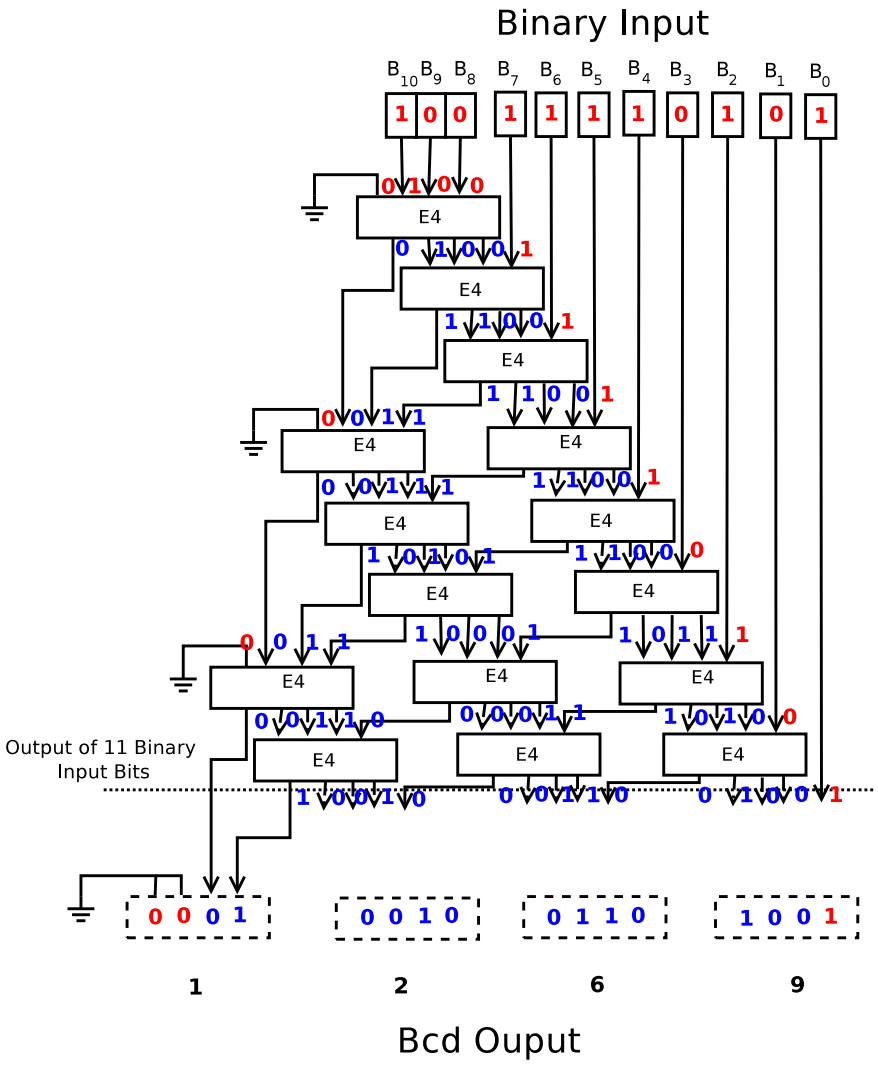


Figure 1.12: 11 Binary bits to BCD decoding tree with E4 decoders numerical example

Line number	Input Code (Octal)	Output Code (octal)
0	00	00
1	01	01
2	02	02
3	03	03
4	04	04
5	05	10
6	06	11
7	07	12
8	10	13
9	11	14
10	12	20
11	13	21
12	14	22
13	15	23
14	16	24
15	17	30
16	20	31
17	21	32
18	22	33
19	23	34
20	24	40
21	25	41
22	26	42
23	27	43
24	30	44
25	31	50
26	32	51
27	33	52
28	34	53
29	35	54
30	36	60
31	37	61
32	40	62
33	41	63
34	42	64
35	43	70
36	44	71
37	45	72
38	46	73
39	47	74

Table 1.3: LookUp Table Content for E6

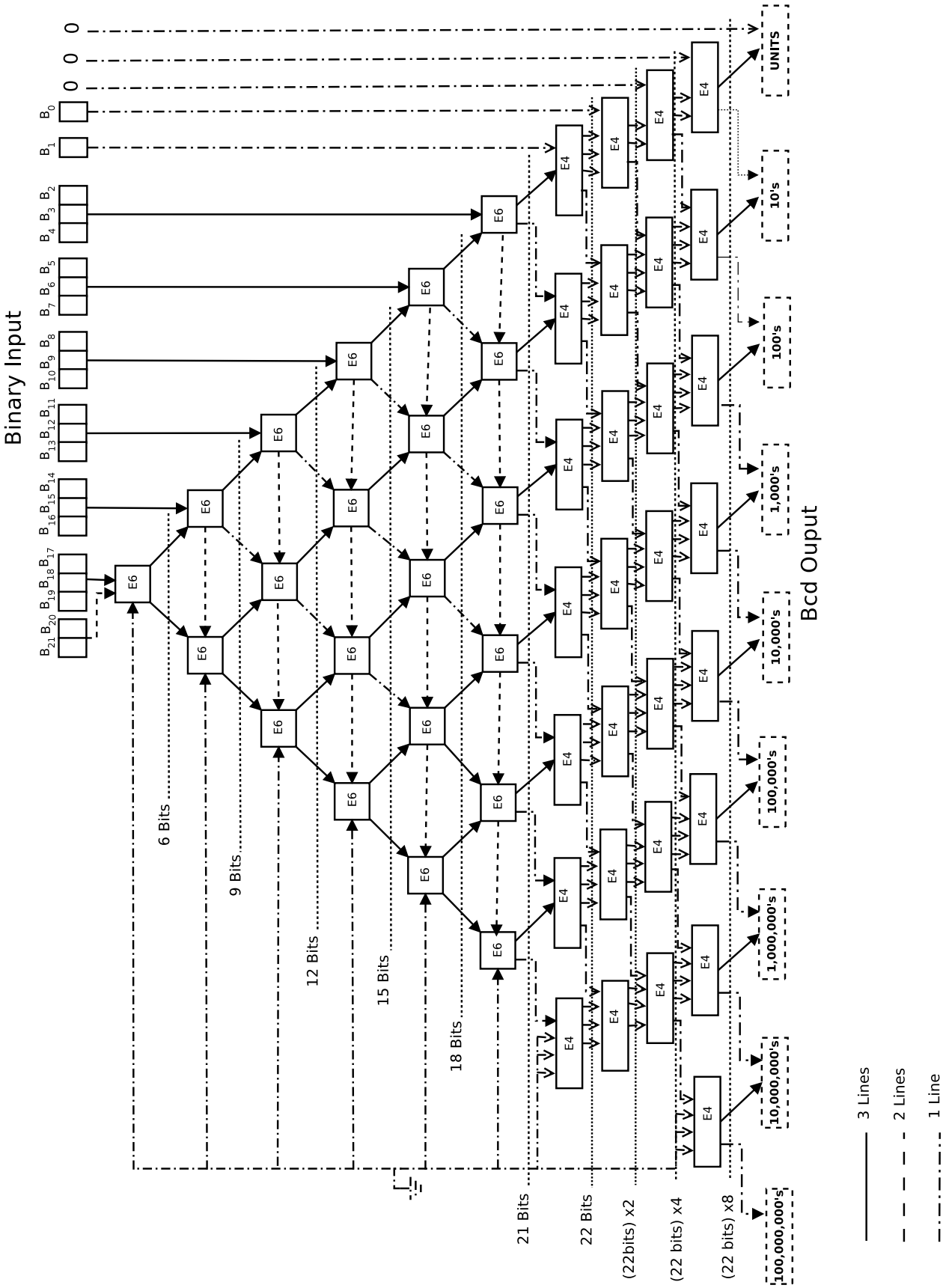


Figure 1.13: Interconnection for E6 static decoders (binary/BCD)

1.4.2.1.2 Algorithms for small binary numbers:

Designs use Couleur's algorithm for small binary numbers: In [41] and [42], the inventor presented different hardware circuits using Couleurs algorithm [14]. In [31], there is an iterative sequential hardware of Couleurs algorithm [14]. Using shift registers and full adders we can get the BCD output vector.

One-step conversion algorithm [34] is another binary to BCD and BCD to binary conversion algorithm. They developed the counting sequence of BIDEC [14] dynamic design to be one-step method instead of two steps in BIDEC dynamic algorithm (Shift and Modify) [14]. For very large number of binary numbers it will consume more iterations (more time) to output the correct conversion data.

A similar way to Couleur's algorithm is presented in [45], the inventor presented a method of converting binary to BCD by shifting and adding the binary number for a certain times. This process is equivalent to multiplying by $(2^{n-1}+1)$.

Three-Four split algorithm and Four-Three split algorithm: Some binary to BCD converters for seven binary bits were described in [7], [27], [28], [46], [9] and [30]. These specific converters are important as they are used to convert the binary partial products resulting from multiplication of two BCD digits to BCD.

The basic cell in [9], [7], [27], [28] and [46] multiplies two BCD digits and outputs the result in seven binary bits. These seven binary bits should be corrected to BCD. Different hardware implementations are proposed for this function. There are constraints for these designs. The binary input size is seven and the output value can not be larger than 81 (in decimal).

The algorithm in [9] converts the seven binary bits to two BCD digits, the steps of this algorithm are:

1. Split the seven binary bits into two groups: the three most significant bits - the four least significant bits.

2. The lower significant group can directly represent a BCD digit if it doesn't exceed $(1001)_2$ or 9_{10}, if it exceeds $(1001)_2$, adding $(0110)_2$ is needed to correct the BCD digits

3. The most three bits have a contribution toward the lower significant digit and the higher significant digit. This contribution is computed to get the two decimal digits as shown in Table 1.4. For example, consider that the most three bits are 011, we can get the decimal value of these bits by adding four zero bits at right so the decimal value of 0110000 is 48, then the contribution to the most significant digit is 4 or "0100" in binary, the contribution to the least significant digit is 8 or "1000" in binary.

Most three bits	Contribution to the most significant digit	Contribution to the least significant digit	Decimal value
000	0000	0000	00
001	0001	0110	16
010	0011	0010	32
011	0100	1000	48
100	0110	0100	64
101	1000	0000	80

Table 1.4: The contribution of the most three binary bits to the two BCD digits

4. A correction is needed to correct the added BCD digits from the previous two steps to form the final two BCD digits.

The authors of [7] continued the work in [9], they have two main contributions:

1. They developed a more efficient architecture to compute the contribution of the two parts in the binary input bits to form the final BCD result.
2. They investigated a different way of splitting the seven binary bits. The new two groups are: Four most significant bits - three least significant bits.

There are two proposed implementations for binary to BCD converter for seven binary bits in [7], one for three-four split algorithm which achieves 15%, 42% improvement over the proposed architecture in [9] and [27] in term of the speed.

Comparing to the implementation of [9], the three-four split algorithm implementation achieves 23% and 17% saving in area and power respectively. The saving in area and power when compared with the architecture of [27] are 39% and 38% respectively.

Converting binary fractions to BCD: Other conversion algorithms are mentioned in [37], [26] and [1]. These algorithms are methods of converting binary and binary fractions to BCD using iterative multiplication/division in [37],[26] and [1].

In [37], the authors proposed a sequential algorithm to convert binary fractions to BCD. The basic idea is to multiply by 10_{10} in each iteration to get the BCD digits.

Consider the binary fraction is expressed as the following equation:

$$f_1 = \frac{d_1}{10} + \frac{d_2}{10} + \frac{d_3}{10} + ...$$

If we multiply the binary fraction number f_1 by 10 in each iteration, we will get a new BCD digit after the sign, the multiplication is done easily by adding the results of multiplying the fraction (f_1) by 2 and 8. This can be done in binary by shifting left one location and three locations. The next step is to add those shifted bits then get the BCD digit for the current iteration. Repeat the previous two steps (Shift and add) to get another BCD digit. Finally we stop when we get the needed number of decimal digits.

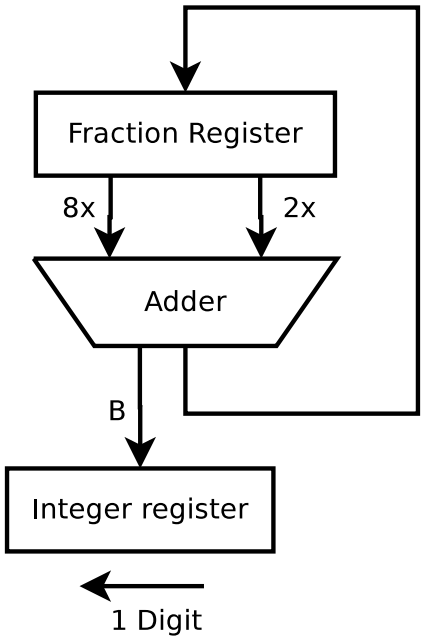


Figure 1.14: The sequential Binary to BCD converter - 1 Digit per cycle

Integer Register (BCD)				Fraction Register (Binary)	Operation
				0.1011	0.6875
				1.011	Times 2
				101.1	Times 8
				110.111	Add
			0110	0.111	Shift integer
				1.11	Times 2
				111.0	Times 8
			0110	1000.11	Add
		0110	1000	0.11	Shift integer
				1.1	Times 2
				110.0	Times 8
		0110	1000	111.1	Add
	0110	1000	0111	0.1	Shift integer
				1.0	Times 2
				100.0	Times 8
	0110	1000	0111	101.0	Add
0110	1000	0111	0101	0.0	Shift integer
6	8	7	5		

Figure 1.15: An example of the algorithm [37]

These methods [37] and [26] are not efficient for large number of binary bits as more we have binary bits as more we take time for more iterations.

Another fractional binary to BCD conversion was presented by R. Fowler [24]. He converted a 24 binary fraction bits into a ten-place decimal using intricate circuitry.

Sequential algorithm using successive multiplication/division operations: Another iterative method [19] for conversion of a binary number to decimal system utilizing division of the binary number by 2 with storage in memory of the reminder, a sequence of operation on the quotient to effect division by 5 so as to obtain at each sequence a number associated with the quotient of successive division by 10 of the number to be converted. The inventor used an adding devices and trigger devices in order to determine the reminders and the partial quotient of successive division by ten.

Another method in [11] divides the N-bit number into 12 bits parts and converts each part into BCD. The method then multiplies at least one BCD number by a variable in response to the number of multiplications to determine the resulting decimal value.

Binary to BCD conversion using bit weights: In [29], [35] and [53], the inventors proposed a method of serial binary input to serial BCD output conversion. The basic idea is using BCD bit weights (1, 2, 4, 8, 10, 20, 40, 80, 100, 200 ...). Summing circuits are used to subtract these weights from the binary number through multiple iterations, at the last iteration we can get the serial BCD number corresponding to the binary input number.

Iterative arrays for radix conversion: Nicoud [32] presents a conversion algorithms from radix "p" numbers to the radix "q" numbers. Four different algorithms for integer and four algorithms for fractions are presented. In our case we can convert from radix 2 to radix 10 and the opposite using these algorithms [32].

For conversion for integers, the authors proposed algorithm that can be implemented as two-dimensional iterative network as shown in figure 1.17, the equation of these cells can be simplified as:

$$a^{+}q + b^{+} = bp + a \tag{1.1}$$

Where a^{+} and a are p-digits and b^{+} and b are q-digits, in our case, p is 2 and q is 10. p-digits $\in = 0, 1$ while q-digits $\in 0, 1, 2, 3, 4, 5, 6, 7, 8, 9$. The basic cell of the two-dimensional iterative network in [32] is shown in figure 1.16.

Figure 1.18 shows three examples for binary to decimal conversion algorithm in [32], the right example shows the hardware implementation to convert "10010110", the first column from the right takes the binary input. "1001" is the most four bits which can be forwarded to the above cell in the first column, according to equation 1.1, the outputs (a^{+}, b^{+}) of this cell will be 1, 9 consecutively. By the same way, we can get the output results of all cells as shown in the example. Finally the BCD digits are 1, 5, 0 which represent

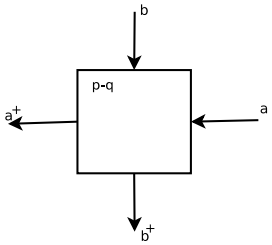


Figure 1.16: The basic cell of the two-dimensional iterative network in [32]

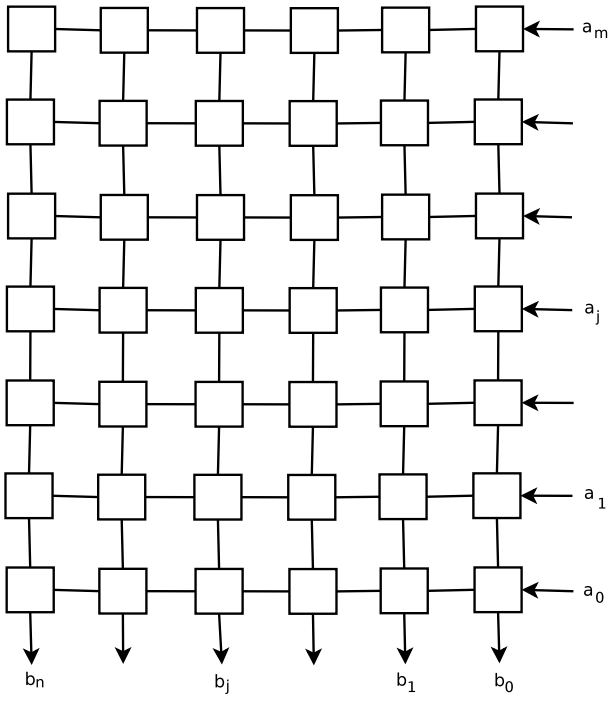


Figure 1.17: Two-dimensional iterative array for conversion of an integer from a radix p to a radix q number system

$(150)_{10}$ in decimal.

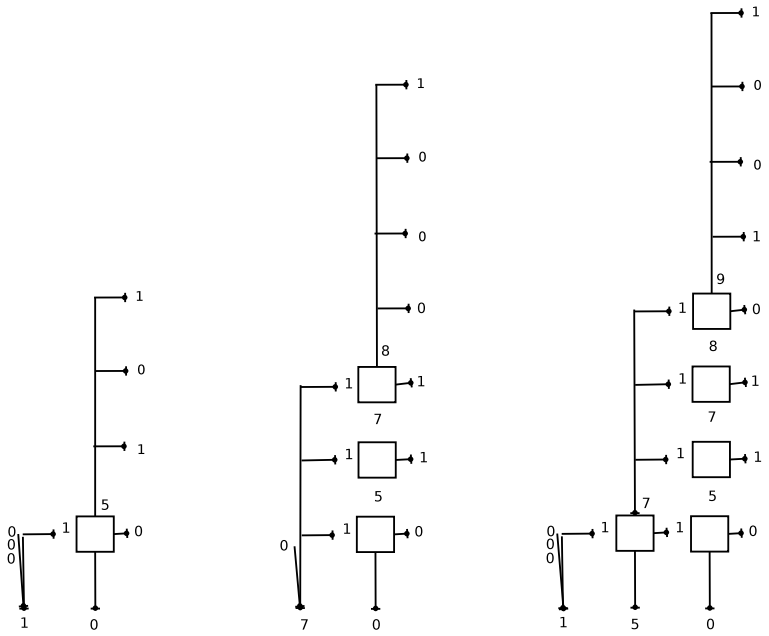


Figure 1.18: An example for binary to decimal conversion algorithm in [32] for integer numbers

For the fraction numbers where the absolute values are pure positive fractional number ($0 \leq N < 1$), The following expressions describe the p-q systems for the fraction numbers:

$$A = \sum_{i=1}^{k} a_{-i} \,.\, p^{-i} \qquad a_{-i} \in P \tag{1.2}$$

$$B = \sum_{j=1}^{l} b_{-j} \,.\, q^{-j} \qquad b_{-i} \in Q \tag{1.3}$$

The values k and l will be chosen in such a way that the maximum error will be negligible.

The main disadvantage of this structure is the large delay for large binary number bits. The delay increases linearly with the number of input bits.

1.4.2.2 BCD to Binary Conversion

1.4.2.2.1 Algorithms for large binary numbers but not efficient: There are different methods of BCD to binary conversions in [14], [34], [32], [25], [23] and [13].

"Shift and Modify" algorithm is used in [14], "one- step" algorithm [34] is a modified algorithm of Couleur's algorithm [14]. A combinational hardware iterative array in [25] that converts from 8421-BCD or 2421-BCD to binary code is one of these methods.

Conversion algorithm from radix 10 to radix 2 [32] is another BCD to binary conversion method.

1.4.2.2.2 Algorithms for small BCD numbers: A Sequential method of BCD to binary conversion was presented in [49] and [13]. Using an iterative addition of the two multiple to the eight multiple of each BCD digit, we can get the resulting binary number.

In [29] and [35], the inventors proposed a method of serial BCD input to binary output conversion. The basic idea is using BCD bit weights (1, 2, 4, 8, 10, 20, 40, 80, 100, 200, ...). If we add the binary numbers of each bit has value '1' in the BCD number we get the corresponding binary number.

In [20], Adder/Subtractor modules in series are used to get the corresponding binary number to the input BCD, This hardware is not efficient in area and delay for large numbers.

In [12], the inventors proposed new decimal to binary conversion method. The proposed hardware receives a BCD number made up of one or more sets of three digits. These three digits is converted to binary then multiplying on 1000 and adding to the sum. At the last iteration we get the binary number.

Using central processing unit and memory chips, BCD to binary conversion method was presented in [10].

1.4.2.2.3 Algorithms have better performance with large BCD numbers: Another method [51] and [23] uses groups of memory chips and levels of binary adders to get the binary output. In this method [51] and [23] large BCD numbers can be converted.

Our design uses new algorithm which divides the BCD input number into digits, then converts each digit with its weight to binary vectors then using CSA "carry save adder" tree we can get the final binary number.

1.4.2.3 BCD to DPD and DPD to BCD conversion

In Our design, we used a block which converts from DPD to BCD and from BCD to DPD [16], [17]. These hardware modules are known as DPD encoder and DPD decoder. DPD encoder divides the BCD number to three digits (12 BCD bits) and converts each set of three decimal digits to 10 bits or 1 declet in DPD (10 bits = 1 declet). DPD decoder converts each set of 10 bits in DPD to 12 BCD bits. The conversion rules of Densely

DPD encoded values										Decimal digits				
b9	b8	b7	b6	b5	b4	b3	b2	b1	b0	d2	d1	d0	Values encoded	Description
a	b	c	d	e	f	0	g	h	i	0abc	0def	0ghi	(0-7) (0-7) (0-7)	Three small digits
a	b	c	d	e	f	1	0	0	i	0abc	0def	100i	(0-7) (0-7) (8-9)	Two small digits, one large
a	b	c	g	h	f	1	0	1	i	0abc	100f	0ghi	(0-7) (8-9) (0-7)	
g	h	c	d	e	f	1	1	0	i	100c	0def	0ghi	(8-9) (0-7) (0-7)	
a	b	c	1	0	f	1	1	1	i	0abc	100f	100i	(0-7) (8-9) (8-9)	One small digit, two large
d	e	c	0	1	f	1	1	1	i	100c	0def	100i	(8-9) (0-7) (8-9)	
g	h	c	0	0	f	1	1	1	i	100c	100f	0ghi	(8-9) (8-9) (0-7)	
x	x	c	1	1	f	1	1	1	i	100c	100f	100i	(8-9) (8-9) (8-9)	Three large digits

Table 1.5: Densely packed decimal encoding rules [15]

Packed Decimal to/from Decimal digits are shown in table 1.5.

In [12], the inventors presented a method for converting scaled binary coded decimal (SBCD) into decimal floating point (DFP) and converting from (DFP) into (SBCD). The coefficient of the DFP number is in DPD.

1.5 Organization of the thesis

In this chapter we presented a brief introduction about the decimal floating point (DFP) addition and the related work to decimal floating point adders and binary to/from BCD converters. Chapter 2 gives an overview of the IEEE754-2008 standard. In chapter 3, we present our proposed decimal floating point adder architecture and implementation. Also we present schulte adder architecture and our implementation for it. Chapter 4 presents the verification of our proposed design and comparison between the synthesis results we get from each design, this chapter includes a conclusion of the thesis and the future work.

Chapter 2: Overview of the IEEE Decimal Floating-Point Standard

As previously indicated, there was an increasing need to DFP arithmetic. Hence, there were many efforts to find out the most appropriate DFP formats, operations and rounding modes that completely define the DFP arithmetic. These efforts ended up with the IEEE 754-2008 floating-point arithmetic standard. This section gives a brief overview to this standard [4].

There are two methods of encoding the significand of DFP numbers in IEEE Std 754-2008 as described before in subsection 1.3, Binary Integer Decimal (BID) [39] and Densely Packed Decimal (DPD). The main difference between the BID encoding and DPD encoding is in the significand representation. In IEEE Std 754-2008, BID encoding is called binary encoding and DPD encoding is called decimal encoding. But either encodings can represent the same DFP numbers.

There are five supported rounding modes in IEEE Std 754-2008 standard. The standard supports four binary formats (binary16, binary32,binary64 and binary128) and three decimal formats (decimal32, decimal64 and decimal128).

2.1 Decimal Formats

The decimal formats are described in subsection 1.3. The DPD encoding represents every three consecutive decimal digits in the decimal significand using 10 bits, and the BID encoding represents the entire decimal significand in binary.

Before being encoded in the combination field, the exponent is first encoded as binary excess code and its bias value depends on the precision used. There are also minimum and maximum representable exponents for each precision. The different parameters for different precision values are presented in table 2.1

Table 2.1: Parameters for different decimal interchange formats

Parameter	Decimal32	Decimal64	Decimal128
Total storage width (bits)	32	64	128
Combination Field (W+5) (bits)	11	13	17
Trailing significand Field (T) (bits)	20	50	110
Total Significand digits (P)	7	16	34
Exponent Bias	101	398	6176
Exponent Width (W) (bits)	8	10	14

In a decimal floating-point format a number might have multiple representations. This set of representations is called the floating-point numbers cohort. For example, if c is a

multiple of 10 and q is less than its maximum allowed value, then (s, q, c) and (s, q+1, c/10) are two representations for the same floating-point number and are members of the same cohort. In other words, a one-digit floating-point number might have up to p different representations while a p-digit floating-point number with no trailing zeros has only one representation (a n-digit floating-point number might have fewer than p-n+1 members in its cohort if it is near the extremes of the formats exponent range). A zero has a much larger cohort: the cohort of +0 contains a representation for each exponent, as does the cohort of -0. This property is added to decimal floating-point to provide results that are matched to the human sense by preserving trailing zeros as discussed before. Hence, different members of a cohort can be distinguished by the decimal-specific operations. In brief, for decimal arithmetic, besides specifying a numerical result, the arithmetic operations also select a member of the results cohort. And thus, decimal applications can make use of the additional information cohorts convey.

2.2 Operations

Operations can be classified as the following:

- **Computational Operations:**
  These operations operate on either floating-point or integer operands and produce floating-point results and/or signal floating-point exceptions.
- **Non-Computational Operations:**
  These operations do not produce floating-point results and do not signal floating-point exceptions. It includes, for example, operations that identify whether a DFP number is negative/positive, finite/infinite, Zero/Non-zero and so on.

Operations can be also classified in a different way according to the relationship between the result format and the operand formats:

- **Homogeneous operations:** in which the floating-point operands and floating- point results are all of the same format.
- **Format Of operations:** which indicates that the format of the result, independent of the formats of the operands.

It should be highlighted that, besides the required operations for a standard compliant implementation, there are other recommended operations for each supported format. These operations mainly include the elementary functions such as sinusoidal and exponential functions and so on.

2.3 Rounding

There are five rounding modes defined in the standard, Round ties to even, Round ties to away, Round toward zero, Round toward positive infinity, and Round toward negative infinity. Also, there are two well-known rounding modes supported in the Java BigDecimal class [33] . Table 2.2 summarizes the different rounding modes with their required action.

Table 2.2: Different Rounding Modes

Rounding Mode	Rounding Mode Rounding Behavior
Round Ties To Away (RA)	Round to nearest number and round ties to nearest away from zero, the result is the one with larger magnitude.
Round Ties to Even (RE)	Round to nearest number and round ties to even, the result is the one with the even least significand digit.
Round Toward Zero (RTZ)	Round always towards zero, the result is the closest DFP number with smaller or equal magnitude.
Round Toward Positive (RPI)	Round always towards positive infinity, the result is the closest DFP number greater than or equal the exact result.
Round Toward Negative (RNI)	Round always towards negative infinity, the result is the closest DFP number smaller than or equal the exact result.
Round Ties to Zero (RZ)	Round to the nearest number and round ties to zero
Round To Away	Round always to nearest away from zero, RA the result is the one with larger or equal magnitude.

2.4 Special numbers and Exceptions

2.4.1 Special numbers

Normal and Subnormal numbers: Operations on DFP numbers may result in either exact or rounded results. However, the standard also specifies two special DFP numbers, infinity and NaN.

A normal number can be defined as a non-zero number in a floating point representation which is within the balanced range supported by a given floating-point format. The magnitude of the smallest normal number in a format is given by $b^{e_{min}}$, where b is the base (radix) of the format and e_{min} is the minimum representable exponent. On the other hand, subnormal numbers fill the underflow gap around zero in floating point arithmetic. Such that any non-zero number - which is smaller than the smallest normal number is subnormal.

Table 2.3: Examples of some DFP operations that invlove infinities, x represents any positive finite non-zero number

Operation	Exception	Operation	Exception
$\infty + x = \infty$	NONE	$\infty / \pm x = \pm\infty$	NONE
$\infty + \infty = \infty$	NONE	$\pm x / \infty = \pm 0$	NONE
$\infty - x = \infty$	NONE	$\infty / \infty = NaN$	INVALID
$\infty - \infty = NaN$	INVALID	$\sqrt{\infty} = \infty$	NONE
$\infty \times \pm x = \pm\infty$	NONE	$\sqrt{-\infty} = NaN$	INVALID
$\infty \times \pm\infty = \pm\infty$	NONE	$\pm x / 0 = \pm\infty$	Division by Zero
$\infty \times 0 = NaN$	INVALID	$remainder(Subnormal, \infty) = NaN$	UnderFlow

Infinities: Infinities represent numbers of arbitrarily large magnitudes, larger than the maximum represented number by the used precision. That is:

$$-\infty < \text{each representable finite number} < +\infty$$

In table 2.3, a list of some arithmetic operations that involve infinities as either operands or results are presented. In this table, the operand x represents any finite non-zero number.

NaNs (Not a Number): Two different kinds of NaN, signaling and quiet, are supported in the standard. Signaling NaNs (sNaNs) represent values for uninitialized variables or missing data samples. Quiet NaNs (qNaNs) result from any invalid operations or operations that involve qNaNs or sNaNs as operands. When encoded, all NaNs have a sign bit and a pattern of bits necessary to identify the encoding as a NaN and which determines its kind (sNaN vs. qNaN). The remaining bits, which are in the trailing significand field, encode the payload, which might contain diagnostic information that either indicates the reason of the NaN or how to handle it. However, the standard specifies a preferred (canonical) representation of the payload of a NaN.

2.4.2 Exceptions

There are five different exceptions which occur when the result of an operation is not the expected floating-point number. The default nonstop exception handling uses a status flag to signal each exception and continues execution, delivering a default result. The IEEE 754-2008 standard defines these five types of exceptions as shown in table 2.4.

Table 2.4: Different Types of Exceptions

Exception	Description	Output
Invalid Operation (Description shows only common examples)	- Computations with sNan operands - Multiplication of $0 \times \infty$ - Effective subtraction of infinities - Square-root of negative operands - Division of $0/0$ or ∞/∞ - Quantize in an insufficient format - Remainder of $x/0$ or ∞/x (x: finite non zero number	Quite NaN
Division by Zero	The divisor of a divide operation is zero and the dividend is a finite non-zero number.	Correctly signed ∞
Overflow	The result of an operation exceeds in magnitude the largest finite number representable.	The largest finite number representable or a signed ∞ according to the rounding direction.
Underflow	The result of a DFP operation in magnitude is below $10^{e_{min}}$ and not zero.	Zero, a subnormal number or$\pm 10^{e_{min}}$ according to rounding mode.
Inexact	The final rounded result is not numerically the same as the exact result (assuming infinite precision).	The rounded result or overflowed result.

Chapter 3: Implementation and Architecture

In this chapter, our proposed design will be discussed and will be compared with one of decimal floating point adder units [43]. Both designs have been implemented on the same hardware target with the same technology. In this chapter we will discuss in details the architecture of each design.

3.1 DFP Addition/Subtraction Technique

In this section, we will describe a high level approach for BID Addition. Consider that we have two DFP operands A, B. A can be represented as (A_{sign}, A_c *and* A_{exp}) and B can be expressed as (B_{sign}, B_c *and*B_{exp}).

At first, the operands may be swapped to enable the simplifying assumption that ($A_{exp} >= B_{exp}$), the swapped operands A_N, B_N, are represented by the triples (A_{Nsign}, A_{Nc} and A_{Nexp}) and (B_{Nsign}, B_{Nc} and B_{Nexp}) respectively.

The addition of two DFP numbers can be performed by aligning the significands. This alignment can be achieved when the exponents are equal but because we have base10 and the significands are in binary form so we need to multiply by powers of 10.

Then we need to add the aligned significands to get the intermediate result , this intermediate result need to be rounded to the format's precision. Rounding DFP numbers by d decimal digits is equivalent of discarding d decimal digits followed by a possible increment of the truncated significand depending on the rounding mode and an increase of the exponent by d, this operation is done in binary so the design of the rounder is not simple as we want to discard d decimal digits in a binary form.

Consider that we have two DFP operands $A = 1234567890123456 \times 10^{17}$,$B = 6543210987654321 \times 10^{14}$. With this technique, A_{Nc} is multiplied by $10^{A_{Nexp} - B_{Nexp}} = 10^3$ to align with B_{Nc}, having exponent of 14. After the addition, the intermediate significand result $Z_{Ic} = 1241111101111110321$ and the intermediate exponent $Z_{Iexp} = B_{Iexp} = 14$. Then the 19-digits intermediate significand is rounded to fit 16 digits ,$d = digits(Z_{Ic}) - p = 19 - 16 = 3$ digits are rounded off and the intermediate exponent is increased by d . $digits(n)$ is a function that computes the number of decimal digits including the trailing zeros in n. In the RTZ rounding mode, the correctly rounded significand and exponent $Z_c = 1241111101111110$ and $Z_{exp} = 14 + 3 = 17$.

3.2 The proposed Adder Architecture

The proposed architecture performs decimal addition on DFP operands represented according IEEE 754-2008 and encoded either in BID or DPD.

The BID operands are first converted to BCD then the BCD numbers enter into a DFP adder and finally the resulting BCD output is converted back to BID.

Alternatively, two DPD operands can be used as inputs and converted to BCD numbers which are fed to the adder. The output of the adder in this case is converted by to DPD. The generic architecture of our proposed adder design is shown in figure 3.1.

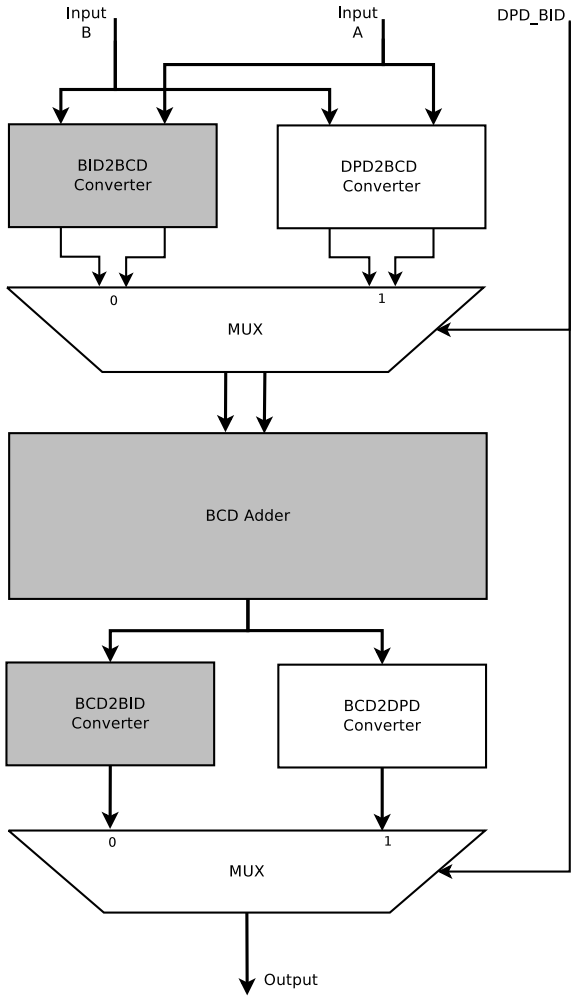


Figure 3.1: Generic architecture of our proposed adder design

In our implementation, we use the SilMinds DFP adder unit which adds two decimal floating point numbers of decimal encoding (DPD) and do all operations in decimal, BID to/from DPD converters to enable getting BID operands and give the result in BID encoding format. In this chapter we will discuss the architecture of our proposed design.

There are two parallel BID to DPD converters for each operands followed by SilMinds DFP adder then DPD to BID converter at the last phase. The block diagram of our proposed design is shown in figure 3.2.

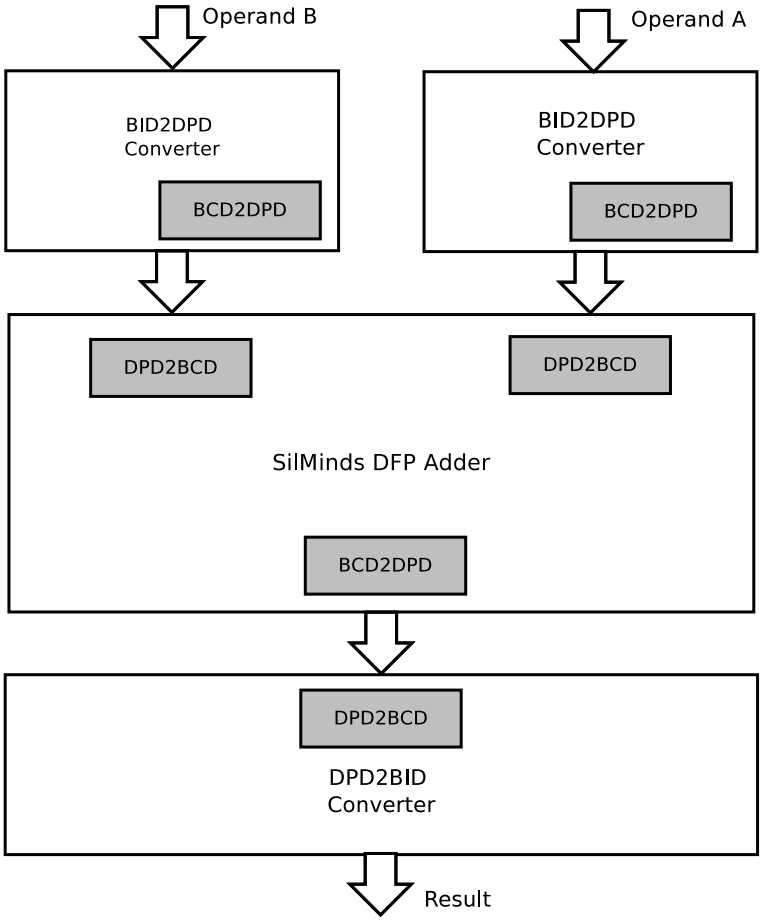


Figure 3.2: Our Proposed DFP Adder Architecture

An enhancement in our implementation can be done if we eliminate the grey blocks in the shown figure 3.2, these grey blocks refer to DPD to BCD converter and BCD to BID converter. The final implementation should contain only the grey blocks in figure 3.1.

3.2.1 SilMinds DFP Adder

This is the main block in our design which does the addition operation. It is the largest area component in the design as shown in table 4.2; this block is owned completely by SilMinds and the company allowed us to use it in this research work.

Based on the exponents values and the number of leading zeros, the significands are aligned. The core of SilMinds adder [22] uses a Kogge-Stone prefix tree adder to implement a fast decimal adder. In this adder, there are two paths; one for data and the other one for carry as shown in figure 3.3. Both the addend and augend are converted to

regular Binary Coded Decimal (BCD) and excess-3 encodings simultaneously. The sum of two BCD digits requires a correction only if it exceeds nine. After the correction of the sum, an incremented BCD digit is produced for each BCD Sum location, the carry signals, which are produced from the other path, select the correct result for each BCD location.

Excess-3 encoding is used in the other path (carry path) to get the propagate and generate signals that are fed into Kogge-Stone tree to get quickly the carry signals corresponding for each BCD digit in the significand.

In parallel, SilMinds DFP adders generate the sticky bit which is used by the alignment shifter in an injection based rounding [47]. SilMinds DFP Adder architecture is shown in figure 3.4.

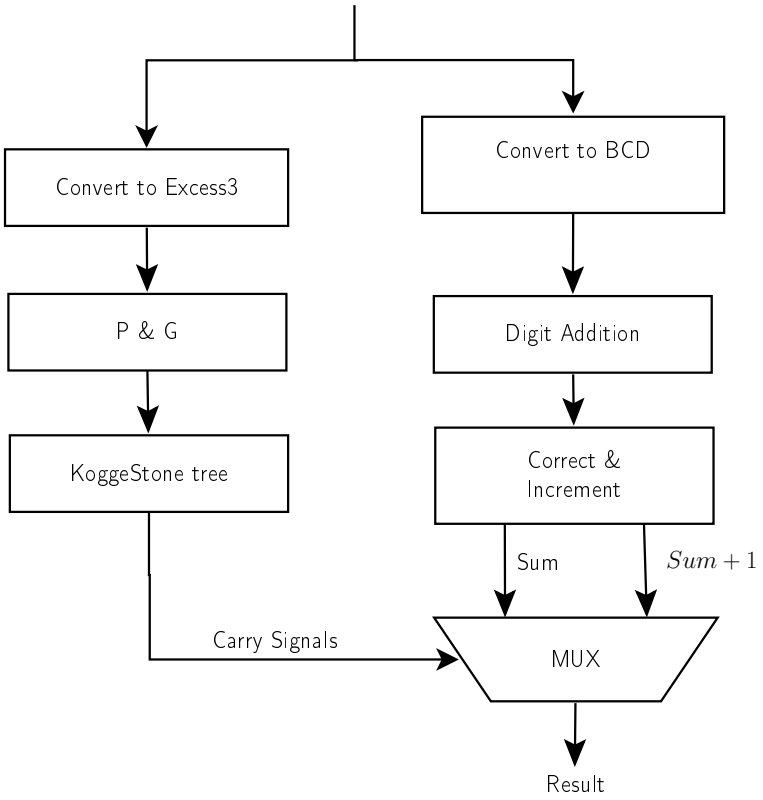


Figure 3.3: SilMinds Fast Multidigit Adder

3.2.1.1 Excess-3 Encoding

Excess-3 is a binary-coded decimal code which simplifies Propagate and Generate P-G signals generation and sum digits correction. It is a complementary BCD code so we can easily detect if the sum of two excess-3 coded numbers is more than 9 or no. This method of using excess-3 encoding simplifies the circuit of decimal addition operation. As shown in table 3.1, it shows the Excess-3 corresponding values for the decimal numbers.

From table 3.1, we can easily detect if the decimal digit is 5 or more when the fourth bit is '1', we can also detect if the sum is 9 by comparing the most significant two bits with "11" and comparing the least two bits by "00". We can check if the sum is more than 9 by

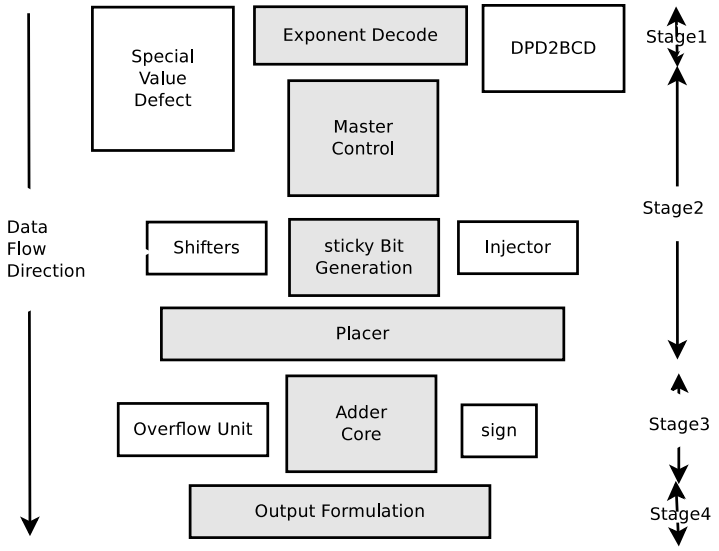


Figure 3.4: SilMinds DFP Adder architecture

Table 3.1: Excess-3 Encoding truth table

Decimal Number	Excess3-Code
0	0011
1	0100
2	0101
3	0110
4	0111
5	1000
6	1001
7	1010
8	1011
9	1100

binary adding of the two numbers in excess-3 encoding and checking the carry signal.

3.2.1.2 Propagate and Generate

SilMinds Adder uses the concepts of generating and propagating carries. The addition of two 1-digit inputs A and B is said to generate if the addition will always carry, regardless of whether there is an input carry (equivalently, regardless of whether any less significant digits in the sum carry). For example, in the decimal addition $52 + 67$, the addition of the tens digits 5 and 6 generates because the result carries to the hundreds digit regardless of whether the ones digit carries (in the example, the ones digit does not carry $(2 + 7 = 9)$).

The addition of two 1-digit inputs A and B is said to propagate if the addition will carry whenever there is an input carry (equivalently, when the next less significant digit in the sum carries). For example, in the decimal addition 37 + 62, the addition of the tens digits 3 and 6 propagate because the result would carry to the hundreds digit if the ones were to carry (which in this example, it does not).

Note that propagate and generate are defined with respect to a single digit of addition and do not depend on any other digits in the sum.

3.2.1.3 Kogge-Stone Tree

The KoggeStone adder is a parallel prefix form carry look-ahead adder. It generates the carry signals in $O(log_n)$ time, and is widely considered the fastest adder design possible. It is the common design for high-performance adders in industry.

As shown in figure 3.5, Each of the shaded circles at the next figure represent the Dot Operator; two P-G pairs are input and a single P-G pair is output to the next stage. It may be modeled as:

$$(P,G) = DotOperator(P_i, G_i, P_{iprev}, G_{iprev}) \tag{3.1}$$

For all shaded circles, the outputs are following these equations:

$$P = P_i \cdot P_{iprev} \tag{3.2}$$
$$G = G_i + P_i \cdot G_{iprev} \tag{3.3}$$

Each of the clear circles at the next figure represent another Dot Operator with a single P-G pair as input and a single P-G pair as output to the next stage. It may be modeled as:

$$P = P_i \tag{3.4}$$
$$G = G_i \tag{3.5}$$

For the square units, A_i and B_i are the inputs of each square unit, each unit has a P-G pair as output, it may be modeled as:

$$P = A_i \oplus B_i \tag{3.6}$$

$$G = A_i B_i \tag{3.7}$$

At the last stage, we get the carry and sum signals according to these equations 3.8 and 3.9:

$$C_i = G_i \tag{3.8}$$

$$Sum_i = P_i \oplus C_{i-1} \tag{3.9}$$

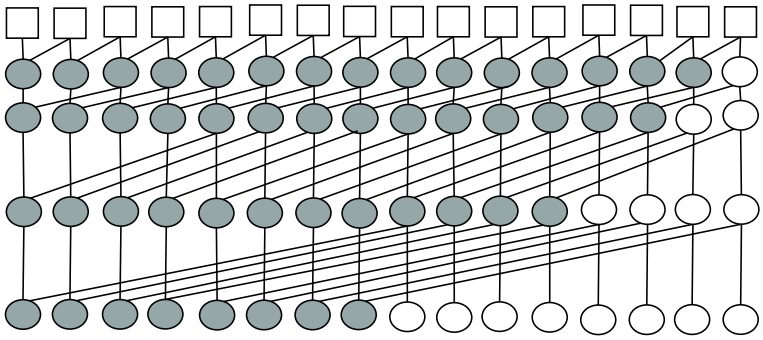


Figure 3.5: The Kogge Stone Adder

3.2.2 DPD to BID Converter

We use SilMinds BID converters in our proposed DFP Adder. As shown in figure 3.2, these units are attached at the beginning and the end to the SilMinds DFP adder unit to set the input/output operands in BID format.

The main aim of this module is to convert DPD encoding [15], [16] and [17] to BID encoding (64 and 128 bits).

Several operations are done in this module:

- Extract the exponent from (G-field).
- Convert the DPD value of T-field to BCD value.
- Extract the whole BCD value from the BCD converted T-field and G-field.
- Convert the BCD value to its equivalent Binary value.

- Decode the most significant bits for the exponent.
- Decode the sign bit.

Where the encoded T-field payload is the Trailing Significand Field where it is decoded to $P-1$ BCD digits, G-field is a combination field that contains the exponent, the most significant digit of the significand, and the encoding classification.

3.2.2.1 System Overview

In this section, a new Densely Packed Decimal Encoding (DPD) (64 and 128 bits) to Binary integer Decimal (BID) Hardware Converter is described.

DPD encoding format [15], [16] and [17] can be divided into four main sections. The first section is the trailing significand field, the second section is the partial exponent field that constitutes the biased exponent, the third section is the combination field which contains the complete other information of the significand (the leading digit),the most two exponent bits and detects if there is a special value while the fourth section is the sign bit.

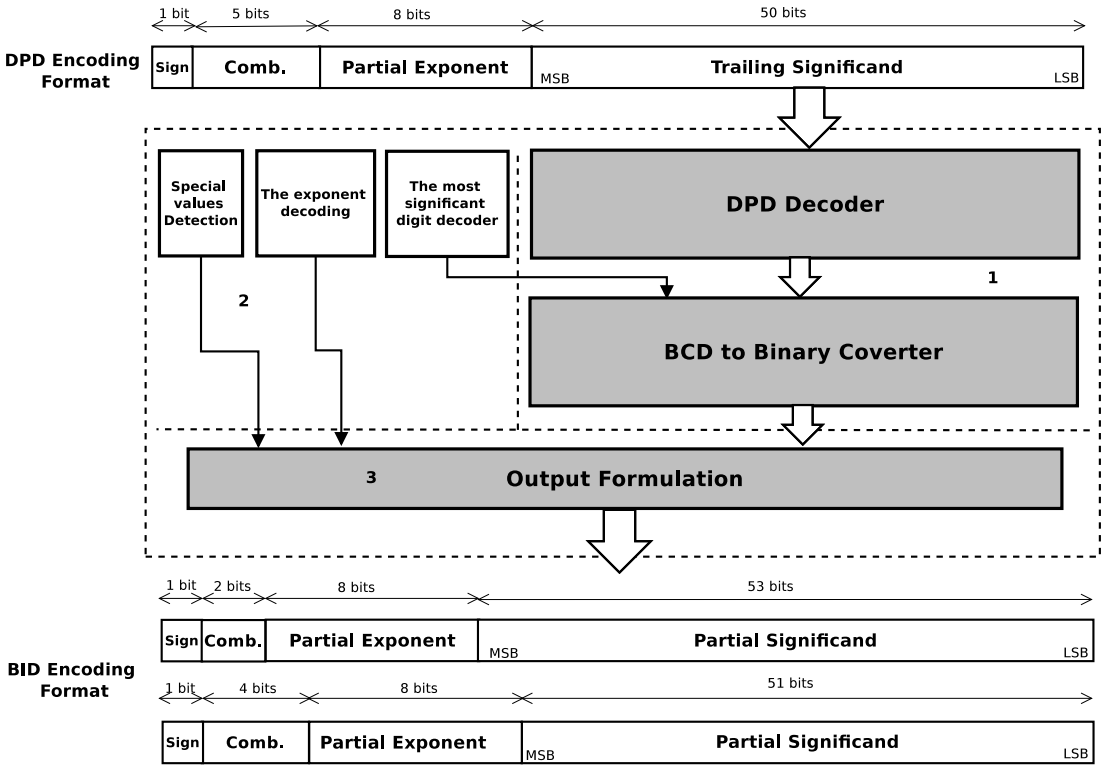


Figure 3.6: DPD to BID converter (64 bits) block diagram

Our design can be divided into three main parts as shown in Figure 3.6 , the first part is the significand conversion from DPD encoding format [15], [16] and [17] to BID encoding format. The second part is some logic units which extract certain fields of BID encoding (the most significant digit, the most exponent bits and the Sign) and the third part is the output block which combines all fields to formulate the output BID encoding.

We differentiate between the special numbers (NaNs and Infinity) and the normal decimal floating point numbers in this part. The grey blocks indicate the critical path delay through it . In this section we will describe each module in the system individually.

3.2.2.2 DPD Decoder

DPD encoding format [15], [16] and [17] is an encoding representation that represents every three consecutive decimal digits in the decimal significand using 10 bits. We can use this module for 64 bits DPD encoding format ([15], [16] and [17]) or 128 bits DPD encoding format [15], [16] and [17] .

In 64 bits DPD encoding format [15], [16] and [17], the DPD trailing significand field size is 50 bits (5 declets 1 declet = 10 bits), each declet can be converted to 3 decimal digit (12 bits in BCD) so 50 DPD trailing significand bits can be converted to 5x12 BCD bits (60 bits).

In 128 bits DPD encoding format, the DPD trailing significand field size is 110 bits (11 declets - one declet size equals 10 bits), each declet can be converted to 3 decimal digit (12 bits in BCD) so 110 DPD trailing significand bits can be converted to 11x12 BCD bits (132 bits).

The conversion from Densely Packed Decimal encoding to BCD are described in table 1.5.

3.2.2.3 BCD to Binary Converter

The main aim of this module is converting BCD number to the binary form.

In 64 bits DPD encoding format, the converted BCD significant size is 60 bits from "DPD Decoder" but there is extended digit, the extended digit has 4 bits size so the overall size of the significand is 64 bits (16 digits).

While in 128 bits DPD encoding format, the converted BCD significant size is 132 bits "from DPD Decoder" but there is extended digit, the extended digit has 4 bits size so the overall size of the significand is 136 bits (34 digits).

Some notes here:

- For DPD64 encoding format the maximum expected decimal value of the completed significand is 9 999 999 999 999 999 (16 digits) which equals 23 86F2 6FC0 FFFF in Hexadecimal (54 bits size), so we have 64 bits width size for the input and 54 bits size for the output of this module. (This module has 64 Input size - 54 output size) as shown in Figure 3.7 .

For DPD128 encoding format, the maximum expected decimal value of the completed significand has size of 136 bits while the converted binary format has maximum size of 114 bits as shown in Figure 3.7.

- All range of the input is not covered as the input is in BCD form where each 4 bits range from 0000 to 1001.

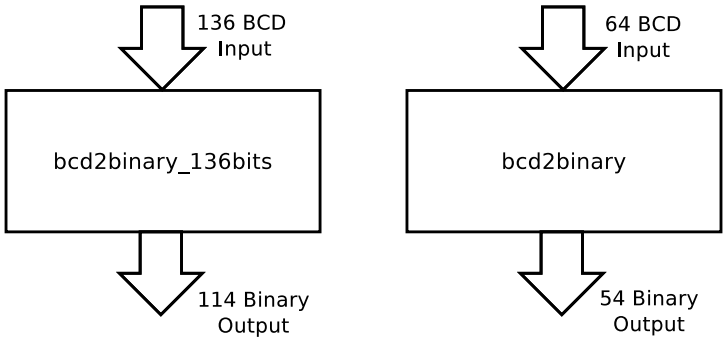


Figure 3.7: BCD to Binary converter entities

The main idea of converting BCD value to binary value is to convert each digit (4 BCD bits) with its weight to the corresponding binary value then we have several(BCD size/4) binary vectors. The sum of these vectors using tree of "three2twocompressor" modules will be the binary output as shown in figure 3.9 for decimal64 format. "three2twocompressor" module adds two binary numbers and outputs the result in Sum/Carry form. We use this module as the critical path delay is small comparable to other adders.

For example, if we have an Input "932" BCD number, we can divide it to three numbers 900, 30 and 2, By adding the binary values of each number we can get the overall binary value.

3.2.2.3.1 Digit to Binary module The important question right now is how can we get the binary values of a decimal number? SilMinds design uses new algorithm which divides the BCD input number into digits, then converts each digit with its weight to binary vectors then using CSA "carry save adder" tree we can get the final binary number.

We can use full truth-table for all possible digits in each weighted locations, for example if we have 3 digits in decimal , the range of these 3 digits is from 000 to 999 and there are 3 weights 1's (units) and 10's (tens) and 100's (hundreds), each weight has ten possible digits from 0 to 9 so we need to save 30 binary numbers which represents the different possibilities we can get.

In our case, for DPD64 numbers, the significand in BCD has 64 bits width size or 16 digits and we need to save 10 binary numbers for each digit so the overall memory size will be (16 digits x 10 possible binary numbers x 54 bits size for each number - 16x10x54 bits or 8,640 bits memory size).

Digit	first component	second component	third component
0	0	0	0
1	1	0	0
2	1sl1	0	0
3	1	1sl1	0
4	1sl2	0	0
5	1	1sl2	0
6	1sl1	1sl2	0
7	1	1sl1	1sl2
8	1sl3	0	0
9	1	1sl3	0

Table 3.2: Decimal Digit Components

A good optimization can be done for the previous solution is to save one value only in each digit weight, the one saved digit is '1' , for example we need to save (1, 10, 100, 1000 and so on till the maximum weight).

We can get it easily from adding shifted versions of the stored binary numbers, as example if we want to get the binary value of 30 000, We have an information of the binary value of 10 000 so if We get the binary value of 20 000 we can add the binary value of 10 000 and 20 000 in binary to get the binary value of 30 000. We can get the binary value of 20 000 by shifting the binary value of 10 000 to the left one location.

Generally, we can get the binary value of the other weighted decimal digit by adding three components as shown in Table 3.2 and Figure 3.8.

sl1 means shift left by 1 bit(multiply by 2), sl2 means shifting left by 2 bits (multiply by 4) and sl3 means shifting left by 3(multiply by 8).

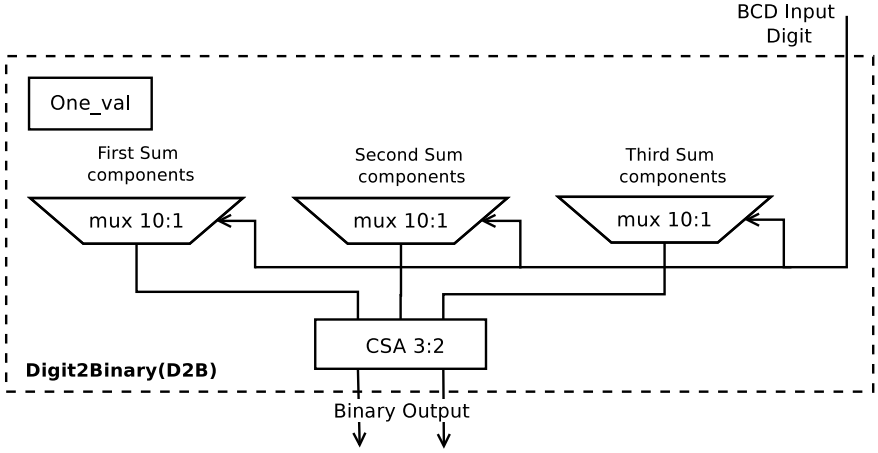


Figure 3.8: Digit to Binary block architecture

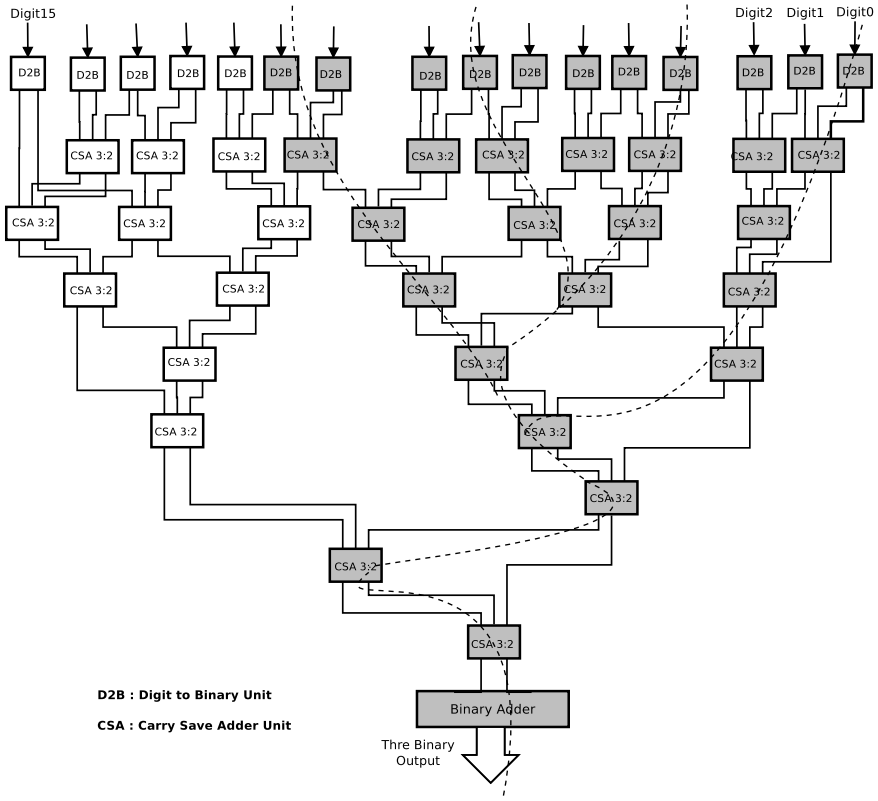


Figure 3.9: BCD to Binary Adder tree - 64 BCD input bits

3.2.2.3.2 The Adder Tree In DPD64 encoding format, we have a BCD number with 64 bits width size (16 digits). We get the least significant 60 bits from DPD decoder but the most significant four bits we extract from the input. In our design we use 16 “digit2binary” modules to convert all digits with its weight to its binary value. As “digit2binary” outputs two vectors, we have 32 binary vectors, the binary output vector is the sum of those resulting 32 binary vectors.

We use “three2twocompressor” modules to add 32 binary vectors through eight levels, at the last stage we have two binary vectors. We use “Binary Adder” module to add two binary vectors at the last stage. The grey modules indicate the critical path delay through it, the critical path may be one of the paths shown in figure 3.9.

3.2.2.4 The most significant digit decoding

This logic is responsible for extracting the most significant digit from DPD64 or DPD128 encoding formats.

As shown in Figure 3.10, 0XXX, 0YYY and 0ZZZ are the extracted BCD digit from DPD64 encoding format, this logic extracts the most significant digit.

But if the most significant digit is 1000 or 1001, 000Y is the extracted BCD digit from DPD64 encoding format as shown in figure 3.11 , this logic extracts the most significant

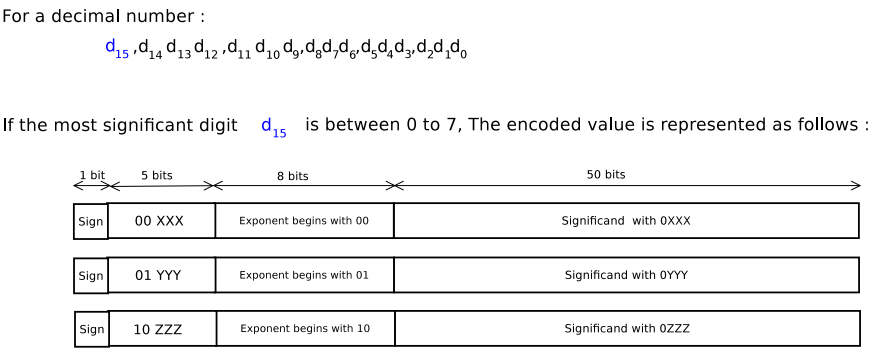


Figure 3.10: DPD64 Encoding format for the most significant digit from 0 – 7

digit. The extracted BCD digit in DPD64 encoding format is concatenated with 60 bits resulting from DPD decoder then the whole 64 bits converted into the binary .

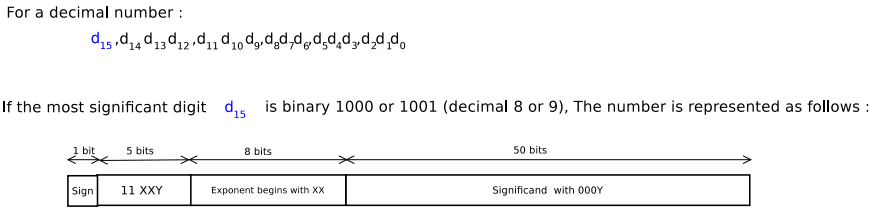


Figure 3.11: DPD64 Encoding format for the most significant digit from 8 – 9

The same idea of extracting the most significant digit in DPD128 encoding format, the extracted BCD digit in DPD128 encoding format is concatenated with 132 bits resulting from DPD decoder then the whole 136 bits converted into the binary form as shown in figure 3.12 and 3.13.

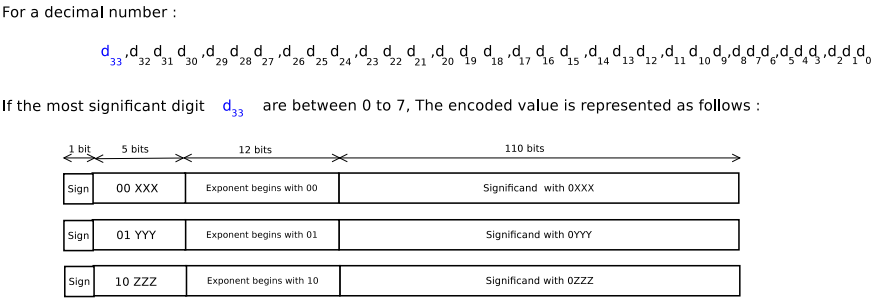


Figure 3.12: DPD128 Encoding format for most significant digit from 0 – 7

3.2.2.5 The most exponent bits decoding

This logic is responsible for extracting the valid most exponent two bits, the exponent bits has a limitations in it. The most exponent two bits are “00,01 or 10”.

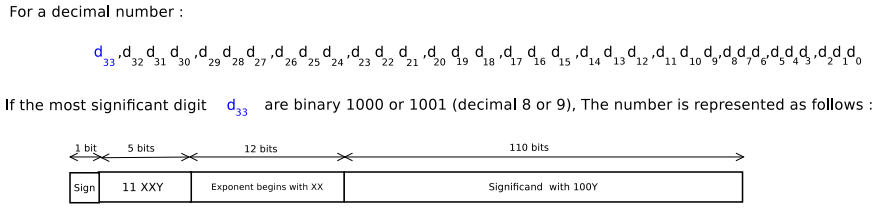


Figure 3.13: DPD128 Encoding format for most significant digit from 8 – 9

In DPD64 encoding format, to get the complete exponent value (10 bits), we should concatenate the extracted most exponent (two bits) with the 8 bits exponent field G5-G12 as shown in figure 3.10 and 3.11.

In DPD128 encoding format, To get the complete exponent value (14 bits), we should concatenate the extracted most exponent (two bits) with the 12 bits exponent field G5-G16 as shown in figure 3.12 and 3.13.

3.2.2.6 Special value detection

This logic is responsible for detecting if the DPD input number is a special value or not. For DPD64 encoding format, the special values has G0-G3 of “1111” (Infinity if G4 = 0 and NaN if G4 = 1). If the DPD input is a special value , BID payload will be Trailing significand field of input(T).

3.2.2.7 The output Formulation

This module combines all field of BID encoding format and outputs it.

3.2.3 BID to DPD Converter

3.2.3.1 System overview

In this design, we convert the binary coefficient in BID encoding format to DPD encoding format through two levels of conversions. Firstly, we convert from the binary form to BCD form then from BCD form to DPD encoding format. We need first to extract the complete binary significand form BID encoding format.

“The significand formulation” module is the module that is responsible of formulating the binary significand, it checks the most significant two bits after the sign bit G0-G1 to formulate the binary significand.

“binary2bcd” converts from binary form to BCD form, in decimal64 the complete binary significand size is 54 bits and the resulting BCD output size from “Binary to BCD” module is 64 bits while in decimal128 the complete binary significand size is 114 bits and

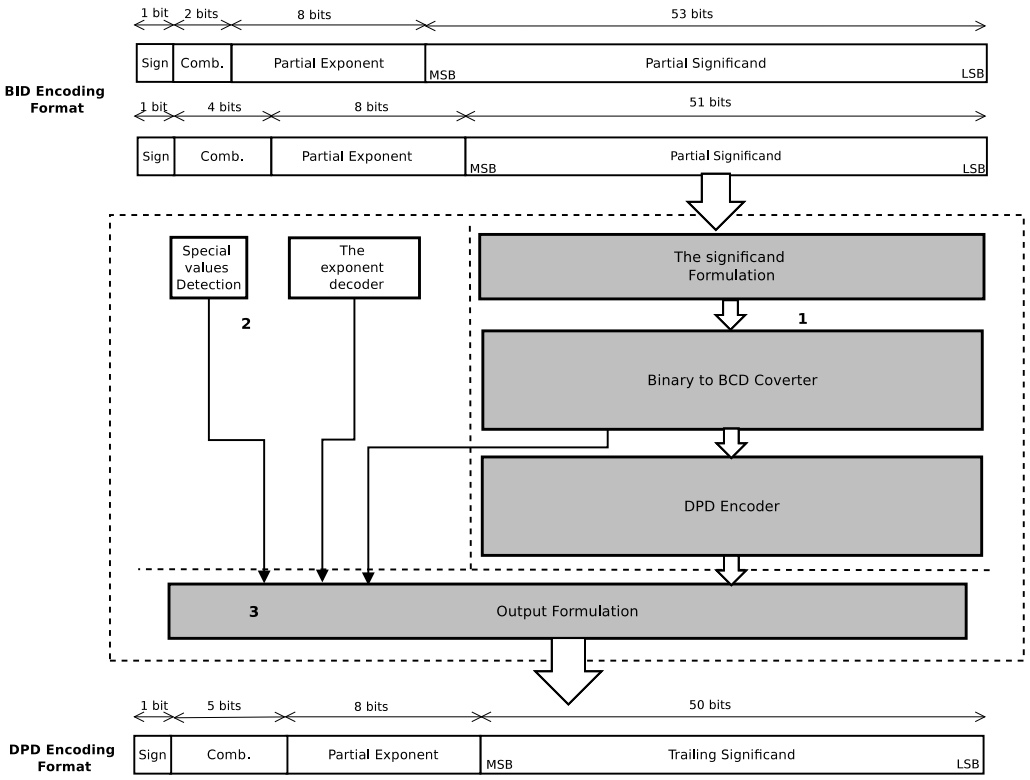


Figure 3.14: Binary Integer Decimal to Densely Packed Decimal Converter block diagram - 64 bits

the resulting BCD output size from “Binary to BCD” module is 132 bits.

There are other logic blocks that decode (the most exponent bits, formulate the DPD combination field and detect the special values (NaNs or Infinity).

The last part of this architecture is the output formulation module, it formulates the DPD encoding format [15], [16] and [17] and outputs it.

3.2.3.2 The significand formulation

In decimal64 , the significand width size can be 53 or 51 bits with hidden ‘0’ , “100” respectively depending on the most significant digit in the decimal number of the coefficient whether its range 0-7 or 8-9. By checking the combination field in BID encoding format we can differentiate between the two forms, if the most significant two bits after the sign bit G0-G1 is “00” or “01” or “10” then the most significant bit of 54 bits significand is ’0’ so the stored bits in the significand field is 53 bits and there is a ’0’ hidden bit at the left so the significand field will be T[52:0], the exponent field will be G0-G9, where the 2 most significant bits (take values of 00,01,10) and the sign bit will be BID[63].

But if G0-G1 is “11” then the most significant bits of 54 bits significand are “100” so the stored bits in the significand field are 51 bits and there are “100” hidden bits at the left so the significand field will be T[50:0], the partial exponent field will be G4-G11,the

combination field will be G0-G3 (G0-G1 = “11”) the sign bit will be BID[63]. Please refer to figure 3.14.

3.2.3.3 The exponent bits decoding unit

This logic is responsible for extracting the exponent bits from BID64 or BID128 encoding formats.

We check G0-G1 at first, if there is “11”, the most significant four bits of the significand are “‘1000” or “1001” - the second BID form for BID64 - as shown in figure 1.5. So the most exponent two bits are G2-G3 and the partial exponent bits are G4-G11.

But if G0-G1 is “00” or “01” or “10”, this mean that the most exponent two bits are G0-G1 and the partial exponent bits are G2-G9 and the most significant four bits of the significand is between “0000” and “0111” - the first BID form for BID64 - as shown in figure 1.4.

The same idea for BID128 as before, if the most two exponent bits after the sign are “11” then the most exponent two bits are G2-G3 and the partial exponent bits are G4-G15 but if G0-G1 are “00” or “01” or “10” so the most exponent two bits are G0-G1 and the partial exponent bits are G2-G13.

3.2.3.4 Special value detection

This logic is responsible for detecting if the BID input number is a special value or not. For BID64 encoding format, the special values has G0-G3 of “1111” (Infinity if G4 = 0 and NaN if G4 = 1). If the BID input is a special value , Nan payload will be the trailing significand filed of the input T.

3.2.3.5 Binary to BCD Converter

SilMinds has three different designs to convert 54-bits binary input, in this section we will describe each design.

3.2.3.5.1 The first design In the first design, we divide the binary input to several parts then convert each binary part using smaller E6 decoding tree (which were described in subsection 1.4.2.1.1) (smaller binary to BCD converter) to several BCD vectors. The next step is to multiply these parts by their corresponding BCD weighted numbers 2^W, where W is the location of the least significant bit in the corresponding binary input part. Using CSA(3:2) (Carry Save adder) tree adder and correction units we can get the BCD number of the binary input.

For example, If we have binary number of 1111 1010 (250 in decimal) then we can divide it into two parts, 1111 (in binary) is the left half part and 1010 (in binary) is the right half part, To get the binary number from these parts we can add 1010 to 1111 multiplied by 2^4. 1010 is 10 in decimal and 1111 is 15 in decimal so the final result will be $10 + 15 \times 2^4 = 10 + 240 = 250$ as expected.

The same idea has been used in our design, dividing the binary number to several parts then converting each part to BCD then multiply the resulting BCD vectors by the corresponding constant binary weighted numbers (2^W) in BCD. Multiplication modules will generate more number of vectors than the number of BCD input vectors. Decimal adder tree is used to add the resulting BCD vectors.

So what are the points we take into consideration when we design?

1. Decrease the number of binary parts as possible to decrease the number of the BCD vectors we need to sum (It will reflect on decreasing the delay of the BCD adder tree).

2. Divide the parts to make the large part size at the left (the most significant bits of the binary number) to decrease the value of the weighted number we multiply on (It will reflect on decreasing the number of BCD vectors so the delay will be decreased).

3. Use binary weighted multiplier number that has decimal digit 0,1,2,4 and 8. More we use these digits in the multiplier number, more we get less vector number.

4. The tree is used in BCD 4221 format so a carry correction is need "mult_carryby2". Arrange the tree units (three2twocompressor and mult_ carryby2 modules) so we can get less delay of the adder.

Using the previous considerations to design 54 bits binary to BCD converter, we divided 54 bits to be as shown in figure 3.15 into three parts: 11 bits, 20 bits and 23 bits.

There are three modules in the design which convert from binary input to BCD output. The first used converter is "bin2bcd_ 11bits", the architecture of this module is something like the shown architecture in figure 3.16 but using 11 input bits and using six E6 unit modules). It converts 11 input bits to (14 bits) BCD output vector.

The second converter is "bin2bcd_ 21bits" which has four BCD output vectors (x1, x2, x4 and x8) (Figure 3.16). There are four output vectors which are (x1, x2, x4 and x8) BCD values of the binary input vector. At the end stages, at least 3 stages of E4 units are used to get more than output vectors of shifted versions of the binary input by one location.

To get (x1, x2, x4 and x8) BCD values, let the input of the tree is [Ip 000] where Ip is the binary input number and it is shifted to the left by three locations.

The expected BCD output value will express x8 of the binary input, to get x4 BCD output take the output from the penultimate stage of binary to BCD tree as shown in figure 3.16, It is the same result if a new tree is used and we use shifted input to the left by two locations [IP 00].

x1 and x2 BCD output vectors take their values from preceding stages for x4 output stage, All output vectors are shown in figure 3.16.

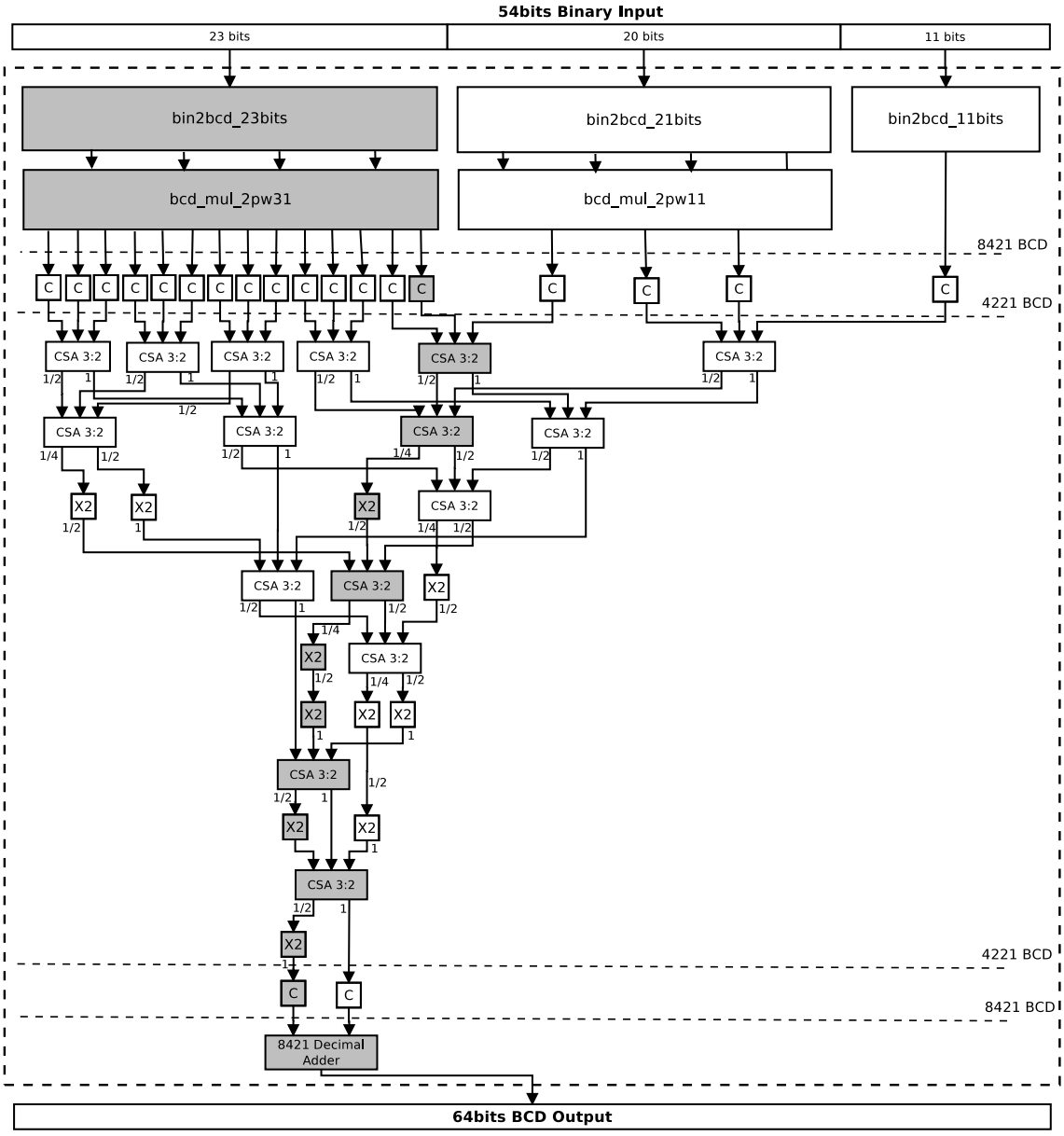


Figure 3.15: 54bits Binary to BCD Converter – The first design

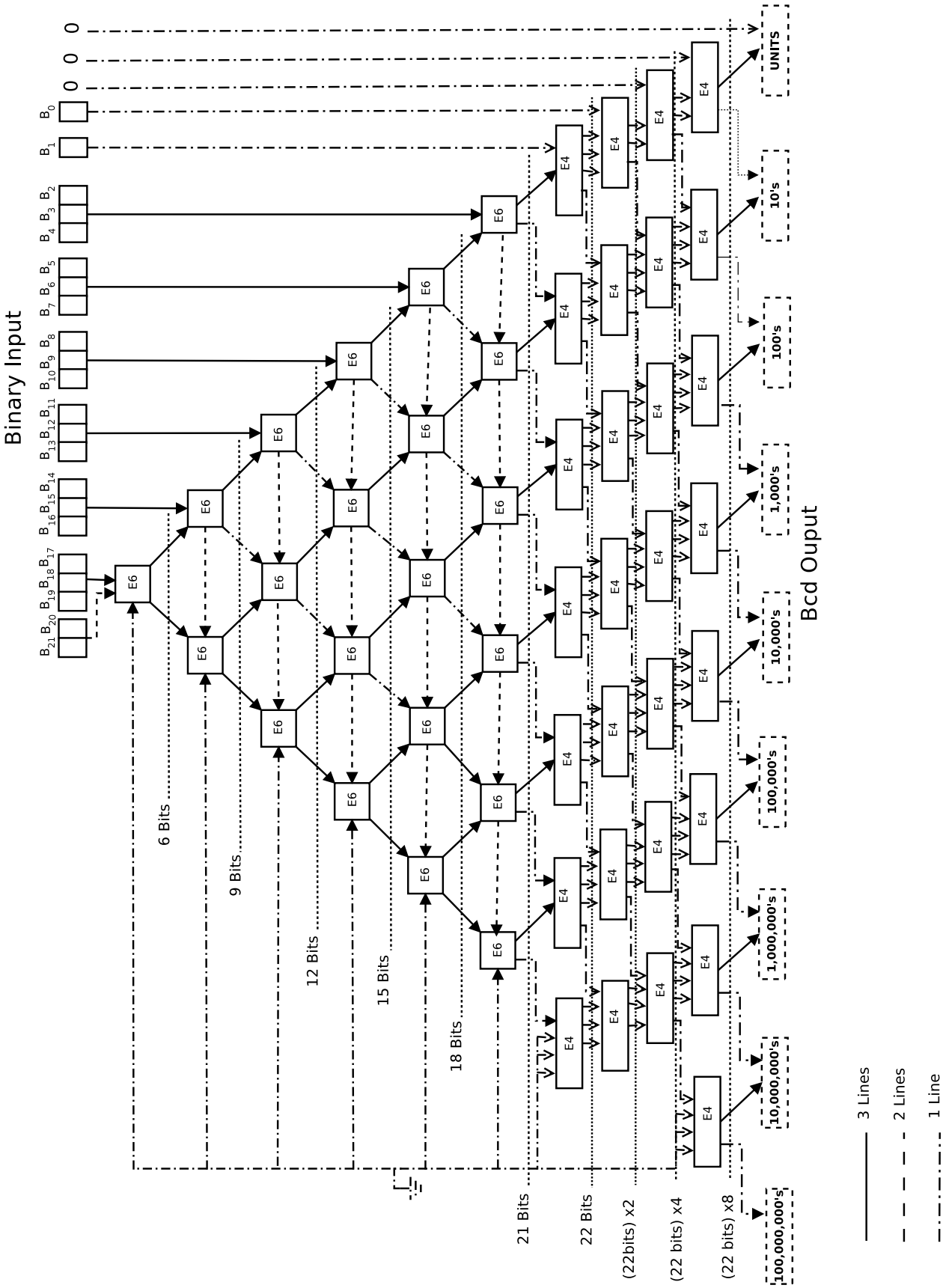


Figure 3.16: Binary to BCD converter – 21 bits

Multiplier digit	Output Vectors	Number of vectors
x1	x1	1
x2	x2	1
x3	x1 x2	2
x4	x4	1
x5	x1 x4	2
x6	x2 x4	2
x7	x1 x2 x4	3
x8	x8	1
x9	x1 x8	2

Table 3.3: The number of vectors for each digit of the multiplier

The input of this module has 21 input size while the output sizes are (30, 31, 33 and 33) bits respectively, the four BCD outputs (x1,x2,x4 and x8) vectors are connected to the BCD multiplier.

The third used converter is "bin2bcd_ 23bits". The input of this module has 23 input size and the four BCD output vectors are connected to the BCD multiplier.

We use an easy technique to implement BCD multipliers with low output delay and low area. This multiplier multiplies by a constant number.

Each BCD multiplier we have has a constant number to multiply, for example "bcd_ mul_ 2pw11" multiplier multiply the BCD input by 2^{11} which is 2048, we can multiply the BCD input by 8, 40 and 2000 individually then output three BCD vectors but if we want to multiply by 2^{20} which is 1 048 576 we can multiply the BCD input vector by 6, 70, 500, 8 000, 40 000 and 1 000 000.

We use shifting to multiply by multiple of ten in BCD and we have four inputs which are (x1, x2, x4 and x8) for the BCD input, we can multiply the input by any decimal digit easily. For example if we want to multiply by 5 , we can produce two output vectors which are x1 and x4 input vectors and if we want to multiply by 7, we can produce three output vectors which are x1, x2 and x4 input vectors. But If we want to multiply by x30 000 we need to generate two vectors (x10 000 and 20 000) using the two techniques above, to get x10 000 we need to shift to the left x1 by four digits and to get x20 000 we need to shift to the left x2 by four digits. for the previous example, If we have a BCD multiplier by 2^{20} or 1 048 576 (in decimal) we need to output these vectors: (x2, x4, x10, x20, x40, x100, x400, x8 000, x40 000 and x1 000 000). The sum of these vectors will be x1 048 567.

Table 3.3 shows the number of vectors we need to output for each digit we need to multiply on.

Table 3.4 shows some parameters for some different binary weighted multiplier numbers 2^W.

W (weight)	The Decimal Number	No. of decimal digits	No. of Non-zero digits	No. of zero digits (no vectors)	No. of multiple of 2 digits (1 vector)	No. of 7 digits (3 vectors)	No. of other digits (2 vectors)	No. of output vectors
9	512	3	3	0	2	0	1	4
10	1 024	4	3	1	3	0	0	3
11	2 048	4	3	1	3	0	0	3
28	268 435 456	9	9	0	4	0	5	14
29	536 870 912	9	8	1	3	1	4	14
30	1 073 741 824	10	9	1	6	2	1	14
31	2 147 483 648	10	10	0	7	1	2	14
32	4 294 967 296	10	10	0	4	1	5	17
33	8 589 934 592	10	10	0	4	0	6	16
34	17 179 869 184	11	11	0	6	2	3	18

Table 3.4: Number of the output vectors from the BCD multiplier for different Binary weighted Multiplier Numbers

As shown in Table 3.4 the grayed rows is the selected BCD multiplier numbers. As it has more number of digits comparable to the near weighted numbers and less number of vectors.

Our proposed design has 3 sets of BCD vectors:

- The resulting BCD vector from “bin2bcd11bits” module (1 vector).
- The resulting BCD vectors from “bcd_ mul_ 2pw11” module (x2048) (3 vectors).
    1. x8 forward x8 Input.
    2. x40 shifting left x40 Input by one digit locations.
    3. x2000 shifting left x2 Input by three digit locations.
- The resulting BCD vectors from “bcd_ mul_ 2pw31” module (x2 147 483 648) (14 vectors).
    1. x8 forward x8 Input.

2. x40 shifting left x4 Input by one digit locations.
3. x600 part1 shifting left x2 Input by two digit locations.
4. x600 part2 shifting left x4 Input by two digit locations.
5. x3000 part1 shifting left x1 Input by three digit locations.
6. x3000 part2 shifting left x2 Input by three digit locations.
7. x80000 shifting left x8 Input by four digit locations.
8. x400000 shifting left x40 Input by five digit locations.
9. x7000000 part1 left shifting x1 Input by six digit locations.
10. x7000000 part2 left shifting x2 Input by six digit locations.
11. x7000000 part3 left shifting x4 Input by six digit locations.
12. x40000000 shifting left x4 Input by seven digit locations.
13. x100000000 shifting left x1 Input by eight digit locations.
14. x2000000000 shifting left x2 Input by nine digit locations.

The total number of BCD vectors which are needed to be added is 18 vector. To get the final result we need to add them.

"Three2TwoCompressor" modules will be used to add BCD numbers but after converting 8421 BCD vectors into 4221 BCD vectors. "Three2TwoCompressor" module has two output vectors, the first one is the sum vector (in weight of unit) and the second one is the carry vector (in weight of half) so in the next sum level we need first to multiply carry by 2 using "mult_carryby2" to add it with unit vectors or to add it directly with other half weight vectors.

Tree of 3:2 compressors and "mult_carryby2" combine the BCD adder tree as shown in figure 3.15, the critical path delay through the grey blocks. It consists of 5 "3:2 compressor" modules and 5 "mult_carryby2" modules.

3.2.3.5.2 The second design The second proposed design is to use complete E6 decoding tree to convert 54 binary input bits to BCD, there are 16 E6 level stages that converts 51 binary bits and there are three E4 stages as shown in figure 3.17.

The advantage of this design that it doesn't use "three_to_two_compressor" or "mul_carry_by2" delays which have large delays but the disadvantage of this design that the tree critical path has large delay.

3.2.3.5.3 The third design In this design, we calculate the BCD value of the 54 binary input bits through adding the BCD values of each binary input bit, We stored the BCD values of 2^0 , 2^1 , 2^2 , 2^3, 2^4 , ... and 2^{53} .

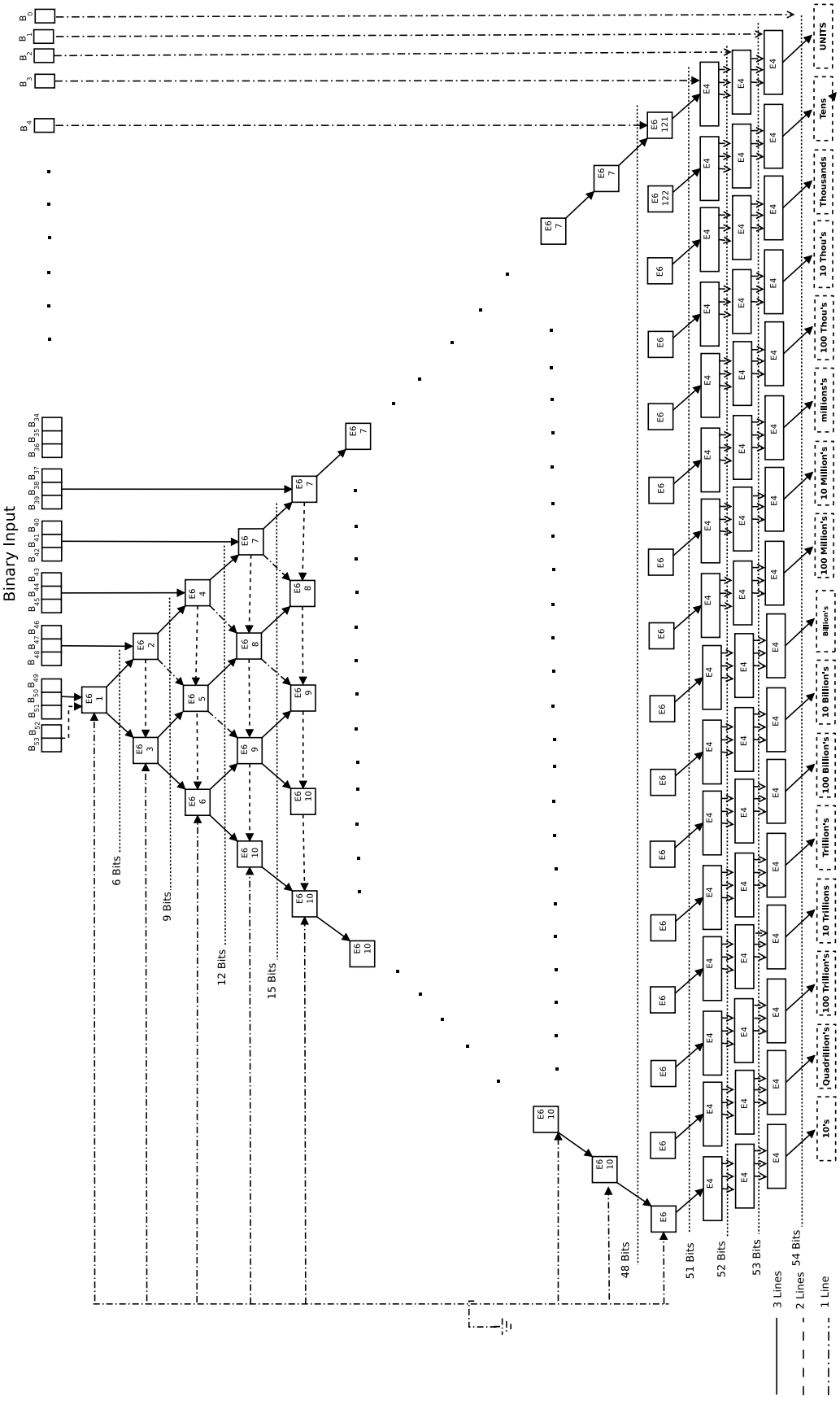


Figure 3.17: 54bits Binary to BCD Converter – The second design

using 54 2x1 multiplexer we output either the corresponding BCD vector value of each bit or zero vector if the corresponding binary input bit equals zero.

54 BCD vectors we get from the multiplexer stage, the resulting BCD vector can be calculated using tree of “3:2 compressor” and “mul_carry_by2” , we use two “bcd_adder_27i_to_4o” modules as shown in figure 3.18. “bcd_adder_27i_to_4o” module is a “CSA” carry save adder tree with correction units which has 27 BCD input and four output vectors.

The advantage of this design that the delay increases logarithmic with the number of binary input bits, so it can be used for large input bits.

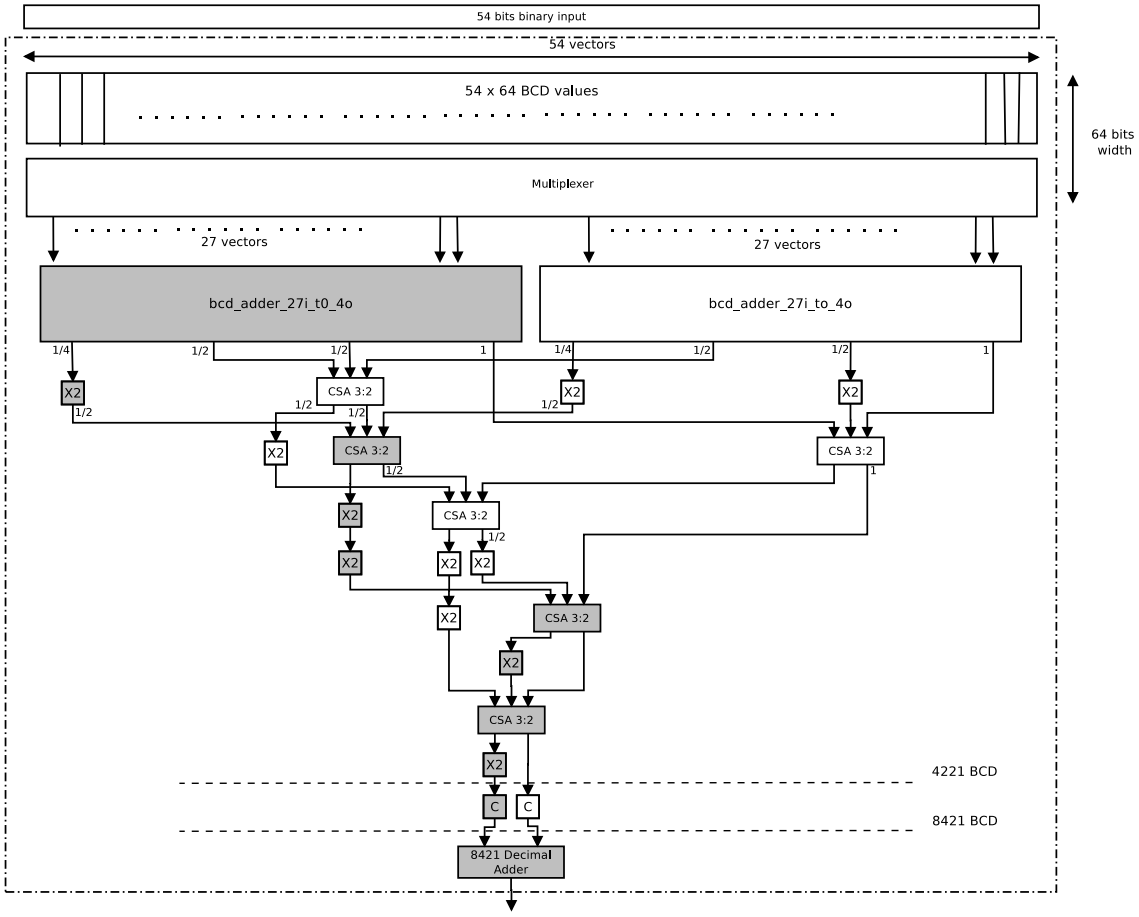


Figure 3.18: 54bits Binary to BCD Converter – The third design

3.3 Schulte Adder

The adder implementation of Schulte, Tsen and Gonzalez-Navarro [43] is the first hardware design for adding and subtracting BID-encoded floating-point numbers. This may be due to a perception that the BID format is more appropriate for software rather than hardware. Contrarily, they argue that BID is well suited for hardware implementations, since it can share hardware with binary arithmetic units. For example, a 64-bit fixed-point multiplier occupies a large percentage of the area of their BID adder, can also be used to perform BFP multiplication, and BID multiplication, comparison, minimum, maximum, quantize, and toIntergalValue.

In this document, we will refer to this implementation as "Schulte Adder". We implemented this adder for decimal64 format according to the adder hardware design architecture which is puplished in this paper[43]. A VHDL code has been written for this design. This design has been implemented on Altera Stratix VI FPGA device family. we estimated the performance and area results for the decimal128 format.

3.3.1 BID Addition/Subtraction Technique in Schulte Adder

Schulte Adder uses the same algorithm described in section 3.1, It uses BID encoding for the DFP numbers. This hardware is not practical for large exponent difference as the maximum exponent difference in decimal64 is 767, this means that the Intermediate result can reach over 2500 bits ($No.of bits = log_2(10^{767}) = 2585$) which is not practical for the rounder architecture as it will take large amount of the hardware resources.

In Schulte Adder they used 64-bits decimal rounder which accept 20 decimal digits ($2^{64} - 1 = 18446744073709551615$) to round it to 16 decimal digits. All decimal rounding operations are in binary format. The problem space now can be divided into three cases. In the first case, Z_{Ic} is guaranteed to fit into the 64-bits rounder as in the previous example. The second case is a special case of the first case, where $A_{exp} = B_{exp}$, this case occurs frequently and doesn't require significand alignment, and requires rounding of at most one digit. In the third case, the intermediate result is too large for the rounder and alternative approach is needed.

3.3.1.1 BID Addition/Subtraction Algorithm

In this section,we will discuss the Schulte Adder algorithm and our implementation for this algorithm. The general algorithm is shown in figure 3.21. The shown algorithm doesn't include the exceptions and special cases for simplicity.

At first, A and B may be swapped to achieve the conditions $A_{exp} \geq B_{exp}$, the swapped operand A_N, B_N are represented by the triples (A_{Nsign}, A_{Nc} and A_{Nexp}) and (B_{NSign}, B_{Nc} and B_{Nexp}).

Then we prepare the operand for further operation, we calculate the difference between the two operand exponents considering that A operand should have the larger exponent

or the same exponent. If not, we need a swap operation to make sure of this condition 3.10:

$$A_{Nexp} \geq B_{Nexp} \tag{3.10}$$

The rest of the circuit is designed by considering this assumption. Also, the effective operation EOP is calculated at this stage as shown in equation 3.11, based on the input operation OP and the signs of the operands. The block diagram of the step 1 is shown in figure 3.19.

$$EOP = OP \oplus A_{Nsign} \oplus B_{Nsign} \tag{3.11}$$

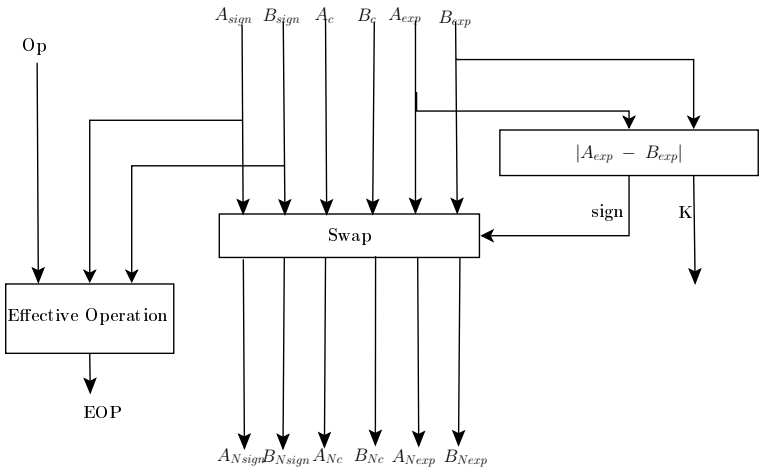


Figure 3.19: Hardware For Algorithm Step 1

The second step is to calculate the number of decimal digits in A_{Nc} which is referred to as Q_a. The number of derived digits has a direct effect on the rounded intermediate result and final result. It will be described in details later.

The next step is to estimate the number of the Intermediate result decimal digits, upon this count we can categorize the output in two main cases:

- The number of Intermediate decimal digits are less than the rounder limit
- The number of Intermediate decimal digits are greater than or equal the rounder limit.

The case of equal exponents is important and it occurs frequently. In this case, we don't need to align the significands so the two significands have 16 decimal digits as maximum.

The intermediate result has 17 decimal digits as maximum so we may need to round this intermediate result one decimal digit. The three cases right now are:

$$Case1: A_{Nc} \times 10^K < 10^{19} \; and \; A_{Nexp} \neq B_{Nexp} \tag{3.12}$$

$$Case2: A_{Nexp} = B_{Nexp}, \; thus \; A_{Nc} \times 10^K < 10^{16} \; as \; K = 0 \tag{3.13}$$

$$Case3: A_{Nc} \times 10^K > 10^{19} \tag{3.14}$$

In the first case, there is no problem for the intermediate result to work with the rounder. The term $(A_{Nc} \times 10^k)$, where k is the absolute difference between the exponnts of A and B ($K = \mid A_{exp} - B_{exp} \mid$), has the larger number of the decimal digits. The maximum possible number of decimal digits is 20 so 10^{19} is selected to be compared to $(A_{Nc} \times 10^k)$ so we add/subtract $A_{Nc} \times 10^K$ and B_{Nc} directly without any preparation. To multiply A_{Nc} by 10^K, we need 64x54 multiplier. A_{Nc} is represented in binary in 54 bits, the binary representation of 10^K is stored in LUT, the maximum value of 10^K is 10^{19} which is represented in 64 bits, this multiplier is described in more details in subsection 3.3.1.2.3. Then we add/subtract the two terms $(A_{Nc} \times 10^k)$ and B_{Nc} and get the absolute value of the intermediate result and compute the sign of the intermediate result.

$$Z_{Isign} = ((10^K \times A_{Nc} \pm B_{Nc}) < 0) \tag{3.15}$$

The operation we do is the effective operation that we determined in the previous step. The inequality operation returns a boolean result '0' or '1', '0' refers to positive sign while '1' refers to negative sign.

The final sign of the result can be computed using this equation:

$$Z_{sign} = A_{Nsign} \oplus Z_{Isign} \tag{3.16}$$

The next step is to round off the intermediate result to fit the 16 decimal digits, as we operate in the first case, we guarantee that the number of decimal digits of the intermediate result will not exceed 19 which will not violate the maximum number of the rounder size. If the intermediate result has size of 16 decimal digits or less, we will not need to round off as the intermediate result fits the 16 decimal digits but if we have more than 16 decimal digits in the intermediate result. We will need to round off this intermediate result by the difference between the number of decimal digits of the intermediate result and the precision size which is 16. The number of digits to round off (d_1) is expressed as:

$$d_1 = max(Q_I - 16, 0), \; where \; Q_I = digits(Z_{Ic}) \tag{3.17}$$

$digits(n)$ is a function that computes the number of decimal digits including the trailing zeros in n.

The final exponent Z_{exp} can be computed using the number of round-off digits and B_{exp} , the number of rounded digits-off d_1 is added to the exponent of the intermediate

result before rounding which is B_{exp} to get the final exponent Z_{exp}.

$$Z_{exp} = B_{exp} + d_1 = A_{exp} - k + d_1 \tag{3.18}$$

In the second case, $A_{exp} = B_{exp}$. This case is an optimized case of the first case. It is so common case so it is worth optimizing.

We don't need significands aligning in this case so we will save a multiply. The rounder can be ignored if the number of decimal digits fits the format's precision. To determine this case, a control signal must be generated if $Z_{Ic} \geq 10^{16}$, if this condition is valid, rounder module is used with one digit rounding off.

The sign of the intermediate result is:

$$Z_{Isign} = ((A_{Nc} \pm B_{Nc}) < 0) \tag{3.19}$$

The operation in 3.19 is the Effective operation (EOP). The inequality operation returns a boolean result '0' or '1', '0' refers to positive sign while '1' refers to negative sign.

The final sign of the result can be computed using the following equation:

$$Z_{sign} = A_{Nsign} \oplus Z_{Isign} \tag{3.20}$$

The final exponent of the result depends on the number of decimal digits of the intermediate result significand. If it is less than 10^{16} then the final exponent is equal A_{exp} and B_{exp} and but if this is not valid, the final exponent can be obtained from incrementing A_{exp}.

Figure 3.20 shows the hardware implementation of the first and the second case. There are two terms, the first term is related to the A_{Nc} which may be multiplied by 10^k , the second term is B_{Nc} in either the first case or in the second case. To differentiate between the first case and the second case, $A_{exp} = B_{exp}$ condition should be checked. If it is achieved, A_{Nc} is selected to be added/subtracted to the second term B_{Nc}. If not, A_{Nc} is multiplied by 10^k, this is implemented using a multiplier and LUT for values of 10^k for K values from 0 to 18.

After generating Z_{Ic}, we need to know the number of digits in the intermediate result. In the first case, the intermediate result may have 20 decimal digits while the standard precision P is 16 decimal digits. The difference between the number of decimal digits in the intermediate result and the standard precision digits is the round-off digits. To get the number of digits in the intermediate result, we need to count it using "decimal digit counter" module.

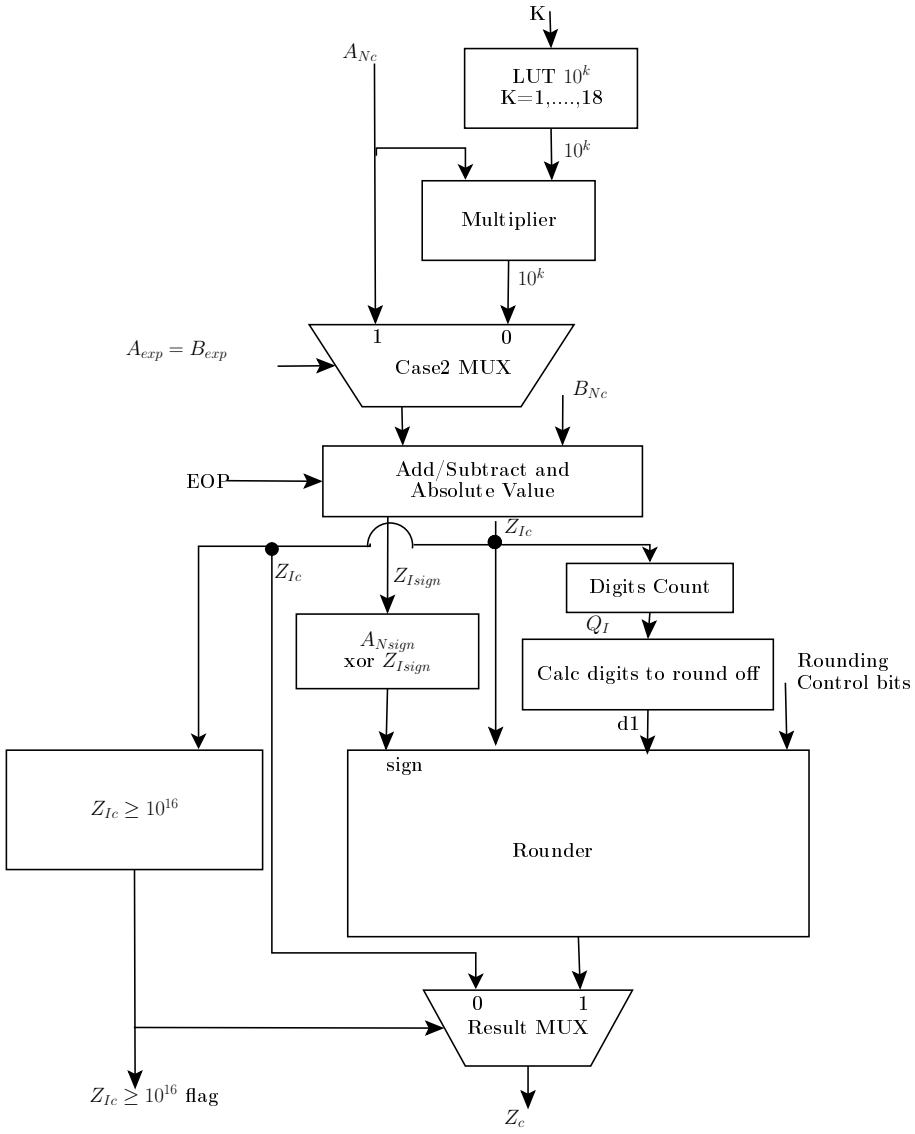


Figure 3.20: Direct Hardware For Cases 1 and 2

In some cases, we don't need the rounder module so a control flag is used to determine if ($Z_{Ic} \geq 10^{16}$). If this flag is set then the rounder must be used. The non-rounded results are bypassed if this condition is not valid.

All operations in the first case and the second case are done in one clock cycle. It is implemented using combinational blocks, no other re-calculations or cycles are needed in the first case and the second case.

The third case handles the condition when the intermediate result is large to be handled by the rounder. In particular, when the condition of $digits(A_{Nc} \times 10^k) > 19$. The block diagram of the third case hardware is shown in figure 3.22. The rounder maximum input size is 20 digits. So what we do is to get maximum possible size of ($A_{Nc} \times 10^k$) but without crossing the standard precision which is 16. The new term will be $A_{Nc} \times 10^g$ Where, $g = 16 - Q_a$, Q_a is the number of decimal digits in A_{Nc}. The second term should be aligned with the first term to have the same exponent. B_{Nc} should be rounded by the difference between K and g so the second term. d_3 is the number of digits in B_{Nc} to be aligned with the first term ($A_{Nc} \times 10^g$). The operation in the equation 3.21 is the effective operation (EOP).

$$Z_{Ic} = A_{Nc} \times 10^g \pm round(B_{Nc}, d_3) \tag{3.21}$$

The intermediate result exponent Z_{Iexp} will compensate the exceeded digits in the first term so it has the value of $A_{Nexp} - g$, Z_{Iexp} is also equal the number of truncated digits from B_{Nc} in the second term so it is $B_{Nexp} - d_3$.

$$Z_{Iexp} = A_{Nexp} - g = B_{Nexp} + d_3 \tag{3.22}$$

The result sign Z_{sign} has the same sign of operand A, A_{Nsign}, as the term which has the larger exponent is the dominant term, the final result has a relative near result of this term.

$$Z_{sign} = A_{sign} \tag{3.23}$$

Although the estimated number of digits is 16, we need to verify that the significand of the intermediate result Z_{Ic} fits exactly 16 digits. Three possible cases after first calculation:

$$10^{16} > Z_{Ic} \geq 10^{15} \tag{3.24}$$

$$Z_{Ic} \geq 10^{16} \tag{3.25}$$

$$Z_{Ic} < 10^{15} \tag{3.26}$$

The first case is the common one which the significand of the intermediate result Z_{Ic} is passed to be the final result Z_c. In the second condition, the significand of the intermediate result has 17 decimal digits, so we need to round-off it by one decimal digit and increment

the intermediate exponent Z_{Iexp} to get the final exponent Z_{exp}.

The final result significand Z_c for the second condition 3.25 can be expressed as:

$$Z_c = round(Z_{Ic}, 1) \tag{3.27}$$

The final result exponent Z_{exp} can be expressed as:

$$Z_{exp} = Z_{Iexp} + 1 \tag{3.28}$$

At the third case, we have another possible digit to fit the result so an adjustment is needed here in our calculation.

The final result significand Z_c for the third condition 3.26 can be expressed as:

$$Z_c = A_{Nc} \times 10^{g+1} \pm round(B_{Nc}, d_3 - 1) \tag{3.29}$$

The final result exponent can be expressed as:

$$Z_{exp} = A_{Nexp} - g - 1 = B_{Nexp} + d_3 - 1 \tag{3.30}$$

$$Z_{exp} = Z_{Iexp} - 1 \tag{3.31}$$

The calculations of the needed adjustments - after verifying the number of digits of the intermediate result if $r > 19$ - is shown in the algorithm flowchart figure 3.21.

As shown in the above equations, We need a hardware to calculate g and d3. g is the needed amount to align with the second term B_{Nc} but without crossing the standard precision which is 16 while d_3 is the number of digits in B_{Nc} to be aligned with the first term $(A_{Nc} \times 10^g)$. A LUT that stores 10^g is needed. $A_{Nc} \times 10^g$ is performed using a binary multiplier. There are more than rounding operations for different signals in the third case. We handle these different cases by determining which cycle we operate in, Some cases need a second cycle if the significand of the intermediate result is greater than or equal 10^{16} or less than 10^{15} .These cases are indicated using a control flag signals. GE10pw16 control signal is used if the significand of the intermediate result is greater than or equal than 10^{16} , Lt10pw15 is another control signal that indicates that the significand of the intermediate result is less than 10^{15}. In both cases, second_cycle signal is set high then other calculations are needed according to the shown algorithm in figure 3.21.

The calculation of second_cycle can be expressed as shown in equation 3.32.

$$second\_cycle \; = \; (GE10pw16 \; AND \; (case2 \; OR \; case3)) \; OR \; (Lt10pw15 \; AND \; case3) \tag{3.32}$$

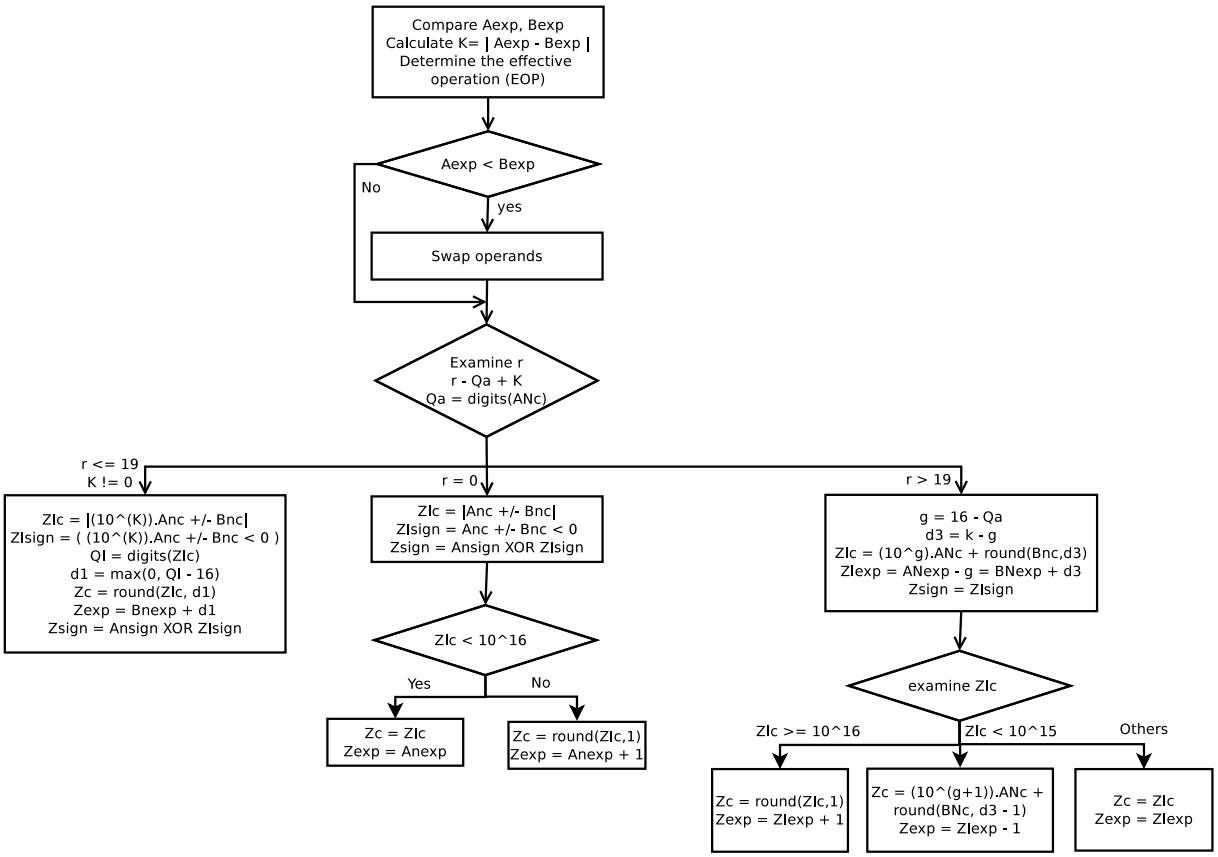


Figure 3.21: The BID Addition/Subtraction Algorithm Flowchart

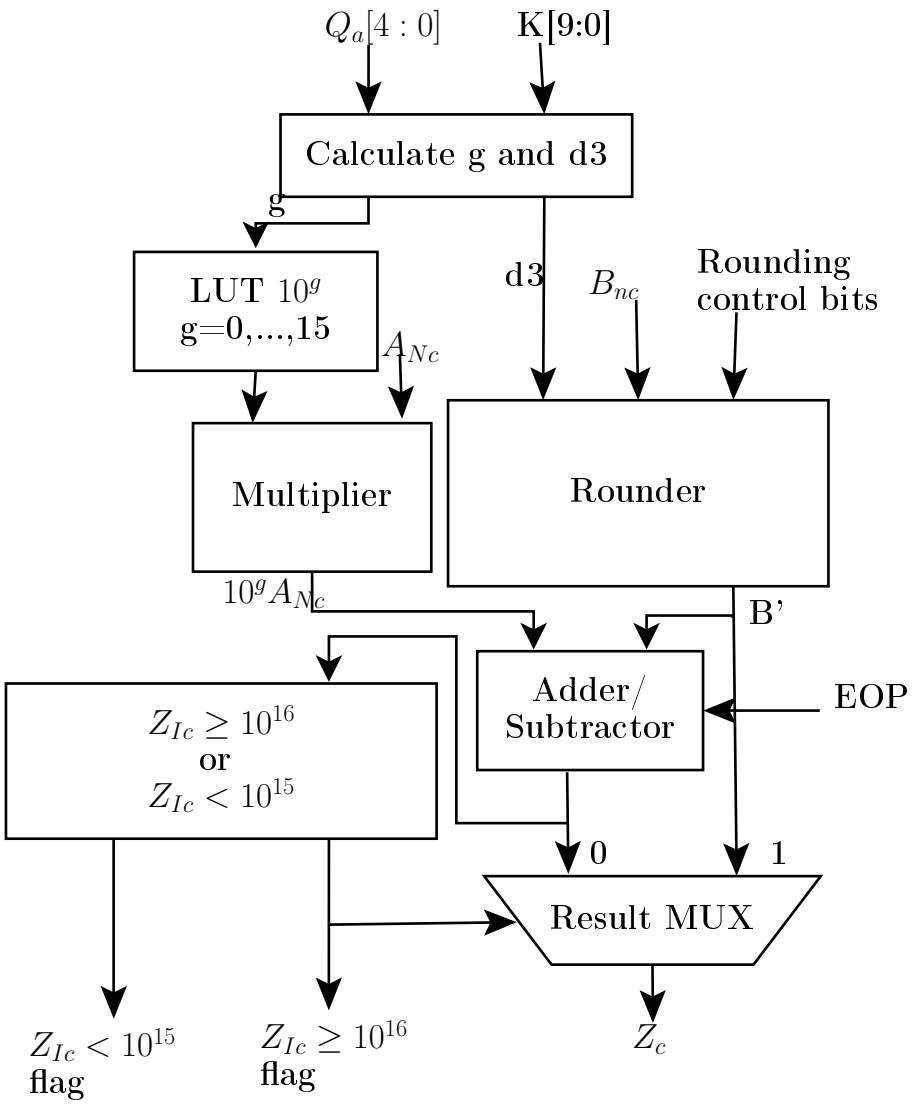


Figure 3.22: Direct Hardware For Case 3

3.3.1.2 The detailed implementation

3.3.1.2.1 Swap module In this module, the two operands A, B are probably swapped depending on the exponents values. The two exponents are compared using 10-bits binary comparator, if the exponent of operand B is larger than the exponent of operand A, Swapping operation is valid.

3.3.1.2.2 Decimal digits counter module In this module, we want to count the number of decimal digits in binary form. We do this operation through more than one step. First, we need to count the number of leading zeros, then we can determine the number of non-leading zeros bits. According to the number of binary bits, We can conclude a first estimation of the number of decimal digits we need. To know the exact number, We compare this binary input value with a certain limit value in the decimal numbers to know if we need to increment the number of expected decimal digits by one or not.

A simple example of that, if we get four binary non-zero bits, we have two probabilites of the number of the decimal digits which may be one or two decimal digits, as the minimum value in four binary bits is 8 which is one decimal digit and the maximum value in the same four binary bits is 15 which is two decimal digits. In this example we need to compare the input with 10 to detect if the input has one or two decimal digits.

Table 3.5 shows the equivalent number of decimal digits for different number of binary bits, for each binary bits, there is corresponding threshold value we need to compare the input with. The limit in table 3.5 depends on the maximum number of binary input bits. We use this module to count the decimal digits for A_{Nc} or Z_{Ic}. The maximum number of binary bits of A_{Nc} is 54 bits while the maximum number of binary bits of Z_{Ic} is 64 bits due to rounder design considerations which will be described later. A top-level design of the decimal digits counter module is shown in figure 3.23.

3.3.1.2.3 64x54 binary multiplier This is a custom binary multiplier for this BID adder. There are two input operands A, B. operand A has 64-bits and operand B has 54-bits. This is a binary multiplier which outputs a 118-bits result.

In this multiplier, We use A as a multiplicand and B as a multiplier, the main idea is to get the partial products of A x B, then using binary adder tree, we can add the partial products to get the final result.

The partial products can be got by multiplying either 0 or 1 by the multiplier, the source of these 0's and 1's is the multiplier. By a simple method, these partial products are shifted with a certain number of positions according to the multiplier bit location then we add these partial products to get the final result.

3.3.1.2.4 Binary Adder Binary adder module is used to add two operands in binary. In our implementation, we used this module for many operations as calculation of the

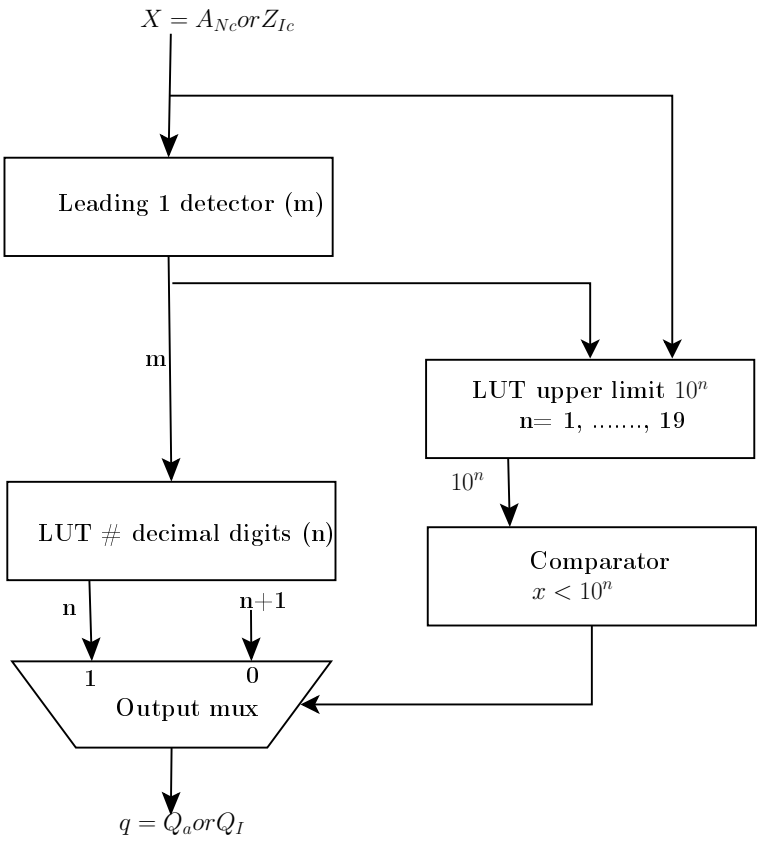


Figure 3.23: Decimal Digit Counter Block Diagram

Table 3.5: The equivalent number of decimal digits for different number of binary bits

The number of binary bits	Threshold value to compare in decimal	No. of Decimal deigits
4	10	1 or 2
7	100	2 or 3
10	1000	3 or 4
14	10000	4 or 5
17	100000	5 or 6
20	1000000	6 or 7
24	10000000	7 or 8
27	100000000	8 or 9
30	1000000000	9 or 10
34	10000000000	10 or 11
37	100000000000	11 or 12
40	1000000000000	12 or 13
44	10000000000000	13 or 14
47	100000000000000	14 or 15
50	1000000000000000	15 or 16
54	10000000000000000	16 or 17
57	100000000000000000	17 or 18
60	1000000000000000000	18 or 19

design parameters and adding the significands of the inputs. Binary adder implementation uses Kogge-stone tree as described before in subsection 3.2.1.3.

3.3.1.2.5 decimal incrementer Decimal Incrementer adds one to an integer number or BCD8421 operand, we use the concept of propagate and generate to get the carry signals for all digits, using these carry signals we can choose between two options, the first option is to get the digit as is while the second option is to get the incremented digit so in this module we have two paths, one for the carry signals and the other path for the data.

To get the incremented digit in decimal we use "digit incrementer" module which increments the decimal digit regardlessof the expected carry. Table 3.6 shows the truth table of the digit incrementer block.

Table 3.6: "Decimal Incrementer" module truth table

A_3	A_2	A_1	A_0	B_3	B_2	B_1	B_0
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	1
0	0	1	1	0	1	0	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	1	1
0	1	1	1	1	0	0	0
1	0	0	0	1	0	0	1
1	0	0	1	0	0	0	0

A number of instances of digit incrementer module increments all decimal digits, this number equals the number of decimal digits or the number of input bits divided by four. Regardless the digits value, the output of the first digit incrementer is always connected to the first decimal digit output of the decimal incrementer module.

For all other digits, there are digit incrementer modules, according to the carry signals values, we choose between the incremented digit and the digit without incrementing.

In our design, we consider that there are two internal operands, the first one is the module's input (to be incremented) and the second operand is a fixed number (0... 001) and the carry in is always '0'. The generate signals (G) will be always 0....01 as we guarantee always that we have a carry generation from the first digit only. The propagate signals (P) represent the possibly digits that may generate a carry, in the BCD8421, the only digit that can generate a carry if there are carry in '1' is '9', the first propagate signal is always '0'.

G	0	0	0	0	.	.	.	0	1
P	X	X	X	X	.	.	.	X	0

For example, if $A = 2499$

1. The first digit is incremented to be 0.
2. The generate signals (G) are 0001
3. The propagate signals (P) are 0010
4. The other incremented digits are 350

According to equations from 3.1 to 3.9, the first three carry bits are 011 and the final output will be 3500. The used kogge-stone tree in this example is shown in figure 3.24.

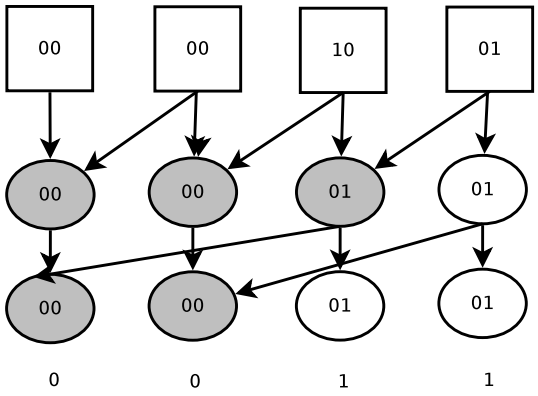


Figure 3.24: An example of decimal incrementer which uses kogge-stone tree

3.3.1.2.6 Binary Rounder The binary rounder unit is an important component of the schulte adder, the implemented rounder is an enhanced unit of other design in a previously published BID rounding unit [44], this enhanced implementation is implemented in Schulte adder and in our implementation for Schulte adder. The new enhanced design supports up to 20 decimal digits or 64 binary bits. In this section we will describe the used rounding algorithm and our implementation.

In this module, the input C_i is truncated d decimal digits, the input C_i and the output (the rounded input) C_o are in binary. An example, consider we have $1234_{10} = 10011010010_2$, if we want to round this number by two decimal digits, the output should be $12_{10} = 1100_2$ or $13_{10} = 1101_2$ depending on the rounding mode. A simple method to do that in binary operations is to divide by 10^d to truncate d decimal digits then to possibly increment the rounded result but his method is costly in terms of area and delay. Instead, the used method is a reciprocal multiplication to avoid division by 10^d and an innovative

approach to ensure that results are correctly rounded in spite of errors due to reciprocal multiplication.

The reciprocal multiplication technique can be viewed as multiplication by precalculated approximation of $W_d \approx 10^{-d}$ to effectively achieve division by $m = 10^d$.

Reciprocal multiplication is well suited for division in cases that the divisors are known and few. In our case we don't have numbers more than 20 decimal digits and the maximum round-off is four as the maximum input for the rounder has 20 decimal digits and it will be fitted in 16 decimal digits. In this implementation we support up to 16 decimal digit round off for 20 input digits (64 binary input).

To get an exact result of the rounded result, we need to get the pre-rounded result which has some information regarding which location it lies. Before or after or on the mid point between two consecutive decimal floating point, or it can be exactly represented as a floating point number with the desired exponent.

To achieve correct rounding when using reciprocal multiplication, the theorem that is published in [44] is applied, where C_i is a u-bit input significand, $m = 10^d$ is a v-bit unsigned integer, w_d is a (u+1)-bit unsigned integer that approximates 10^{-d}, left shifted by u+v bits, and P is the (2u+1)-bit product of $C_i \times W_d$. P is partitioned into three fields, Q, R, and D as shown in figure 3.25, where Q gives the truncated quotient (i.e, floor($C_i/10^d$)), R provides rounding information, and D is discarded.

Based on this theorem in [44], Q provides the truncated quotient, $q = floor(C_i/10^d)$, and R provides all information needed to determine the correctly rounded result. We elaborate on how this theorem helps to detect midpoints and exactness by focusing on the calculation of the two parameters u and v. The number of bits in the input significand, , is referred to as u. In our implementation , u is fixed at 64 bits, since the maximum input, $10^{20} - 1$, fits within 64 bits. The number of bits needed to represent 10^d, is referred to as v, where

$$v = ceil(log_2(10^d)) = ceil(d.log_2(10)) \tag{3.33}$$

For example,10^5 is represented in 17 bits, so v is 17 when rounding off d = 5 decimal digits. It is important to note that v varies with d as shown in table 3.7, whereas u is fixed.

The (2u+1)-bit product, $P = C_i W_d$ is shown in figure 3.25, where P is partitioned into three fields: Q, R, and D. D consists of the u least significant product bits, which contain no useful information and are discarded. The Q and R fields occupy the remaining (u+1)-bits and vary in width depending on v. The most significant (u+1-v) bits, Q, correspond to the truncated quotient. The next lowest v bits, R, are inspected to determine if the truncated decimal digits, when viewed as a fraction, represent exactly zero, exactly one half, between zero and one half, or above one half.

To determine whether the pre-rounded result is exact, less than a midpoint, a midpoint, or greater than a midpoint, we use the properties described in table 3.8. The round bit, r*, is the most significant bit of the R field, and the sticky bit, s*, is set if any of the remaining bits of the R field are 1. As shown in table 3.8, these two bits keep enough information to

Table 3.7: v versus d

d	10^d	v (No. of bits of 10^d)
1	10^1	4
2	10^2	7
3	10^3	11
4	10^4	14
5	10^5	17
6	10^6	20
7	10^7	24
8	10^8	27
9	10^9	30
10	10^{10}	34
11	10^{11}	37
12	10^{12}	40
13	10^{13}	44
14	10^{14}	47
15	10^{15}	50
16	10^{16}	54

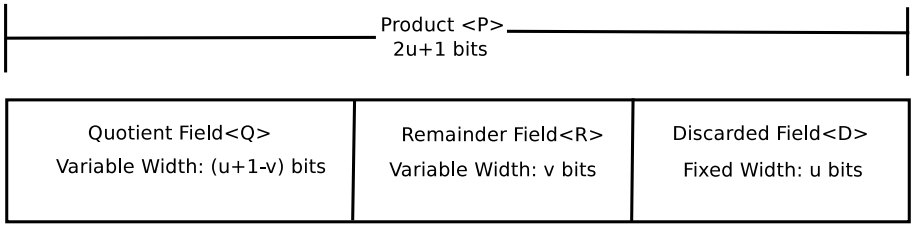


Figure 3.25: Product Fields

know if the pre-rounded result is exact (r*=0, s*=0), less than a midpoint (r*=0, s*=1), a midpoint (r*=1, s*=0), or greater than a midpoint (r*=1, s*= 1).

Table 3.8: Midpoint and Exact Results

Conditions	Values of R	r*	s*
Pre-rounded result is exact	$R == 0$	0	0
Pre-rounded result less than midpoint	$R < 2^{v-1}$	0	1
Pre-rounded result exactly midpoint	$R == 2^{v-1}$	1	0
Pre-rounded result greater than midpoint	$R > 2^{v-1}$	1	1

In our implementation, we pre-calculate sixteen values of and store them in a lookup table. The lookup table has the input d , it is considered as the address of the different stored entries. The pre-calculated values of W_d used in this design are shown in table 3.9. Since u=64, each table entry is u+1= 65 bits. Though 65 bits are needed for the entries each values most significant bit is always 1. Thus, only 64 bits are stored per entry, and the most significant bit of W_d is hardwired to 1. A value of W_d is selected from the table 3.9, based on the number of digits to round off. The 64-bit input C_i , is then multiplied by to produce the result P, a 129-bit intermediate value.

Table 3.9: Pre-calculated values of W_d

$I/m = 10^{-d}$	LUT Location	W_d
10^{-1}	0001	65'h1999999999999999A
10^{-2}	0010	65'h147AE147AE147AE14
10^{-3}	0011	65'h10624DD2F1A9FBEFF
10^{-4}	0100	65'h1A36E2EB1C432CA58
10^{-5}	0101	65'h14F8B588E368F0846
10^{-6}	0110	65'h10C6F7A0B5ED8D36B
10^{-7}	0111	65'h1AD7F29ABCAF48578
10^{-8}	1000	65'h15798EE2308C399FA
10^{-9}	1001	65'h112E0BE826D694B2E
10^{-10}	1010	65'h1B7CDFD9D7BDBAB7D
10^{-11}	1011	65'h15FD7FE17864955FE
10^{-12}	1100	65'h119799812DEA11198
10^{-13}	1101	65'h1C25C268497681C26
10^{-14}	1110	65'h16849B86A12B81C26
10^{-15}	1111	65'h1203AF9EE756159B2
10^{-16}	0000	65'h1CD2B297D889BC2B7

The top-level design of our rounding unit is presented in Figure 3.26. The design takes as inputs, the 64-bit input,C_i , an 11-bit value, d, that indicates the number of digits to be rounded off, the sign of the input operand,S_i , and the 3-bit Rounding Mode. The output is a 54-bit rounded significand, C_o.

Figure 3.26 shows the block diagram of the rounder design, the middle and left sides of this diagram shows the data path of this module. In the right side, there is logic control

unit which generate some control signals.

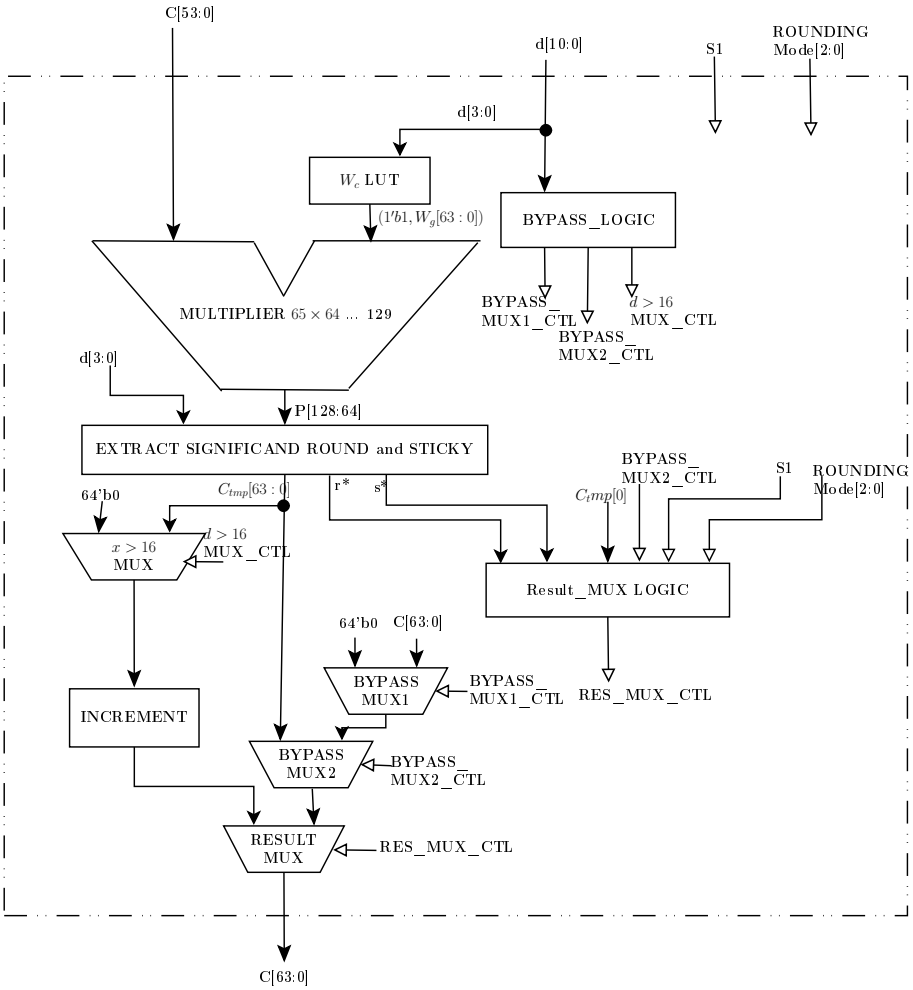


Figure 3.26: Schulte Rounder Block Diagram

RES_MUX_CTL is a control signal which choose between the incremented path and the normal path without incrementing. This control signals is dependent on the rounding mode. This implementation supports five rounding modes in the IEEE standard (754-2008):

- roundTiesToEven (RTE): rounds to the nearest value; if the number falls midway it is rounded to the nearest value with an even (zero) least significant bit, which occurs 50floating-point and the recommended default for decimal.
- roundTiesToAway (RTA): rounds to the nearest value; if the number falls midway it is rounded to the nearest value above (for positive numbers) or below (for negative numbers); this is intended as an option for decimal floating point.
- roundTowardZero (RTZ): directed rounding towards zero
- roundTowardNegative (RTN): directed rounding towards negative infinity
- roundTowardPositive (RTP): directed rounding towards positive infinity

dGt16_MUC_CTL is the control signal which select between the pre-rounded result C_{tmp} from the "Extract Significand" module and zeros, zeros are selected in the case of d is greater than 20 which is the rounder maximum limit.

BYPASS_MUX2_CTL selects between the normal path C_{tmp} if there is valid rounding operation or another path as the number of digits rounded off d is zero, or special values.

BYPASS_MUX1_CTL choose between C_i if no rounding is needed (d = 0) and between zreo in case of special values as NaNs,$\pm\infty$, these values can not be rounded.

The extract significant, round and sticky bit unit following 65x64 multiplier, is used to extract Q, r* and s* fields. P[128:64] is the important part in the pre- rounded result, we can get the rounding bit r* by shifting right this part by v-1, r* is in its least significant bit location while Q is the most u+1-v significant bits. This unit extracts also the sticky bits which uses a mask obtained from a lookup table on d[3:0] to keep only relevant bits of P[128:64]. It then OR's these all bits to get the sticky bit s*.

The table 3.10 shows when *RES_MUX_CTL* control signal is activated, based on the sign, r*, s*, $C_{tmp}[0]$ and the rounding mode.

Table 3.10: Logic Equations for Increment Control

Rounding Mode	Increment Condition
roundTiesToEven	r* & (s* — $C_{tmp}[0]$)
roundTiesToAway	r*
roundTowardZero	0
roundTowardPositive	S_i & (r* — s*)
roundTowardNegative	S_i & (r* — s*)

An example of the rounder unit, consider we have an input 6721500 and we want to round-off three digits in roundTiesToEven (RTE) rounding mode, from table 3.10, the precalculated W_d for d = 3 is $10624D2F1A9FBEFF_{16}$ which multiplied by the input 6721500_{10}, given a product $P = 690600000000036929424_{16}$. Since $v = 10$ and $u = 64$ the quotient $Q = P[128:74] = 00000000001A41_{16} = 6721_{10}$, $C_{tmp} = 00000000001A41_{16}$ and the remider $R = P[73:64] = 200_{16}$, this means that r* = 1 and s* = 0. Which means that this pre-rounded result is an exact mid-point. In RTE, the increment condition for the mid-point numbers is $C_tmp[0] = 1$, this condition is achieved so the increment operation is valid for this number, the expected result is 6722_{10}

3.3.1.2.7 Our Enhancement in Binary Rounder A new design is presented here for the binary rounder, the design has better results in term of area and delay when compared to above rounder of schulte [44]. The synthesis results are shown in 4.4.

The main idea is to convert 64-bits in binary to 20 decimal digits represented in BCD, then to do our rounding which can be translated to shifting and some small logics, then to determine if an increment operation is needed,then to convert to binary form.

As shown in figure 3.27, First we convert the 64 bits bianry input C_o to BCD format using BIN2BCD converter module, the input has size of 64 bits while the output has 80 bits size. Then a decimal shifter which takes the number of digits to be rounded off d and shift right by $4 * d$ bit locations. The most significant four truncated bits represent the rounded digit R while the others truncated bits represent the sticky bits.

At this time, we can determine easily in decimal representation if the rounded digit is greater than or equal 5 using simple hardware. In the control unit we get the rounded bit r* and s*, r* can be expressed as:

r* = R[3] OR (R[2] AND (R[1] OR R[0]))
s* = ORing(all sticky bits)

The control unit also checks the increment condition as shown in table 3.10, it outputs the increment flag to select between the incremented rounded number and the non-incremented rounded number.

The selected signal is converted to binary using BCD2BIN module which converts from BCD format to binary format. This converted output is the final rounded result C_o

In this design, the most area and delay are consumed in the converters.

3.3.1.3 Combined BID Adder Design

Cases 1-3 have been presented to explain the ideas behind the BID adder design. Figure 3.28 shows complete schulte BID adder design, which shares components to use less hardware, by adding multiplexers and control logic. With intelligent scheduling and adding an additional path through the rounder, only one multiplier is needed. The multiplication used to push A_{Nc} to the full decimal64 precision of p = 16 digits is performed with the multiplier inside the rounder. To simplify figure 3.28, the multiplier is darkened to illustrate this point.

The ability to reuse the multiplier in the DFP unit has important effects. First, by adding logic around the multiplier, many functions may be incrementally added to a DFP solution at a modest area cost. Second, the high utilization of the multiplier requires sophisticated scheduling with several possibilities for optimization including adding buffers, reservation stations, and control logic. These details are not shown in Figure 3.28.

3.4 Conclusion

In this chapter, we propose a novel decimal floating point adder, the proposed architecture performs decimal addition on DFP operands represented according to IEEE 754-2008 and

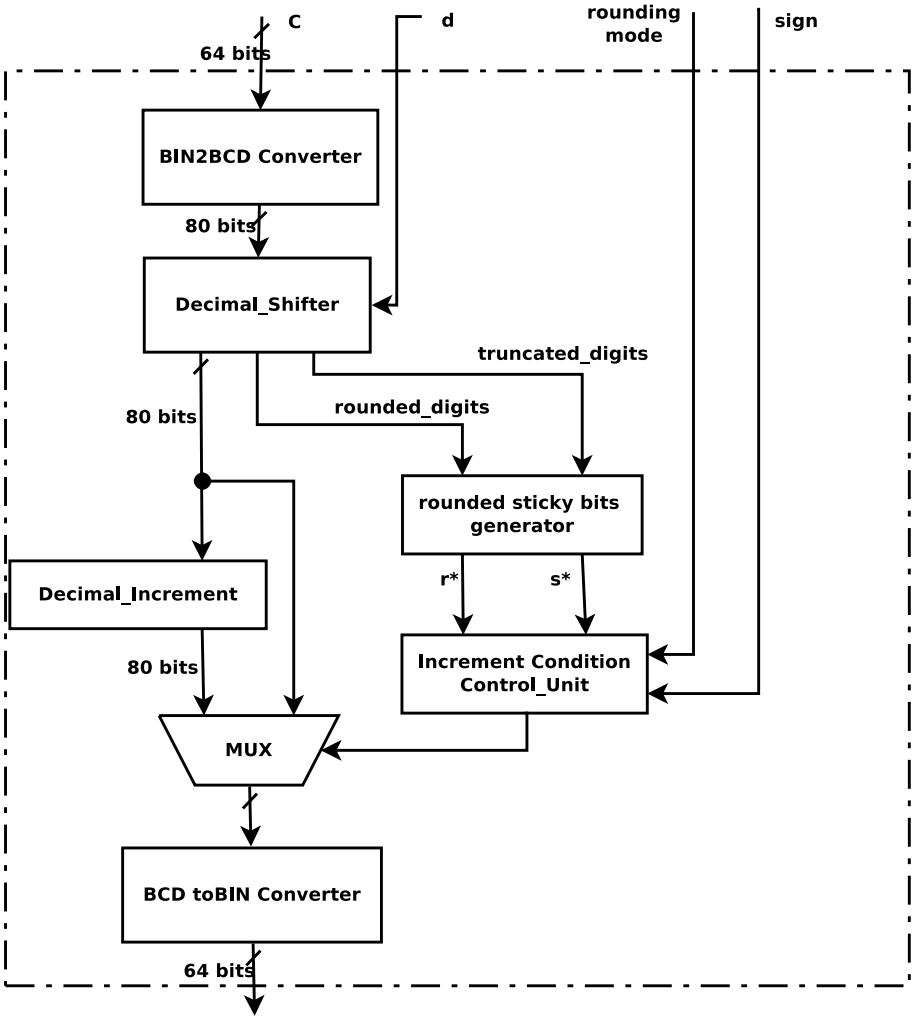


Figure 3.27: The Proposed Rounder Block Diagram

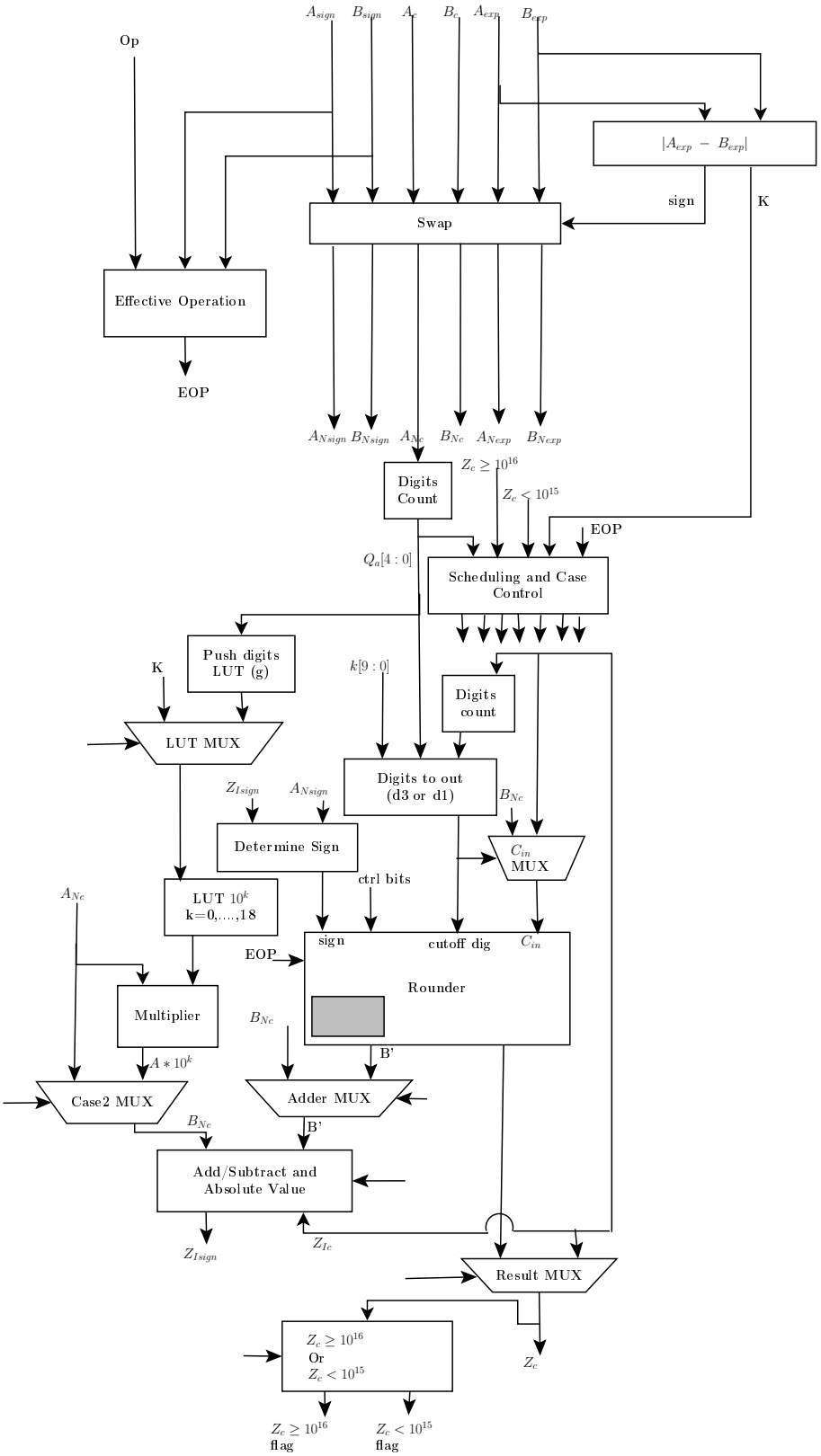


Figure 3.28: Adder HW Design Combined with Case 1, 2 and 3

encoded in BID.

We also propose to use novel converters from BID to DPD and vice-versa to surround the inner core of SilMinds DPD adder design [22]. This conversion from BID followed by the core of a DPD unit and finally a conversion back to BID is proved in Chapter 4 to be a better alternative to direct BID units' implementations [43].

In this chapter, we propose a new binary rounder design which is used in the BID adder units' implementation in [43].

Chapter 4: Experimental Results

In this chapter, we present the verification and the synthesis results of the proposed implementation and compare it with the other implemented design. At the end of this chapter, we suggest some points as a future work to extend or enhance our proposal in this thesis.

4.1 Functional Verification

The verification for decimal floating point units is very interesting research topic as the test space is large. For our adder unit, there are two operands and five rounding modes and two operations (addition subtraction), the total test space is $2^{2X64} \times 5 \times 2 \cong 3.4 \times 10^{39}$ test vectors. The average simulation time on a system using Intel $Core^{TM}$ 2 Duo 2 GHz processor and 3 GB RAM is 6.7 msec/test vector. The time needed to test the design on all possible test vectors would then be 7.23×10^{29} years. Hence, we used only a much smaller number of testcases. The total number of the tested vectors is 268200. Some of test vectors are generated by Cairo University [36] and IBM [6]. For all test vectors, both implemented designs are passed successfully.

4.2 Synthesis Results

The designs were synthesized using Altera Quartus II tool on Stratix VI FPGA family, EP4SGX230KF40C2 device. The Optimization technique for analysis and synthesis is set to "Balanced" which means balance high performance with minimal logic usage. The following table 4.1 show the synthesis results of the implemented decimal floating point adders (the proposed design and the Schulte Adder design - without any modifications in the binary rounder design).

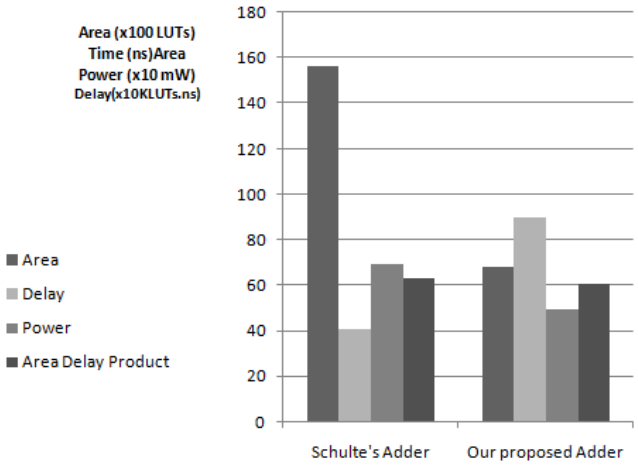
Table 4.1: synthesis results of the implemented decimal floating point adders (the proposed design and the Schulte Adder design - decimal64)

	Area (LUTs)	Delay (ns)	Power (mW)	Area Delay Product (KLUT.ns)
The proposed Adder design	6787	89.46	484	607.165
Schulte's Adder design	15638	40.18	647	628.335

The results in table 4.1 show 56.6% reduction in the area and 25.2% reduction in power consumption and 3.4% reduction in Area Delay Product compared to the Schulte's BID adder. Power consumption and Area Delay Product results are better for the proposed adder despite the increase in the time.

The column chart in figure 4.1 compares between the proposed adder design and Schulte's adder for Area, Delay, Power and Area Delay Product for decimal64 mode.

Figure 4.1: Comparison between the synthesis results of the proposed Adder and Schulte Adder - decimal64



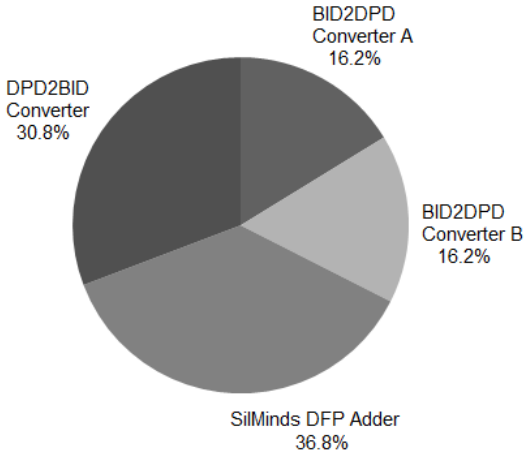
Our proposed adder design is divided into three main parts, the synthesis results are shown in table 4.2.

Table 4.2: The synthesis results of the proposed Adder - decimal64

	Area (LUTs)	Delay (ns)
BID2DPD Converter A	1100	40.5
BID2DPD Converter B	1100	40.5
SilMinds DFP Adder	2491	29.5
DPD2BID Converter	2085	19.5

The following chart 4.2 shows the area profiling for our proposed Adder in decimal64 mode.

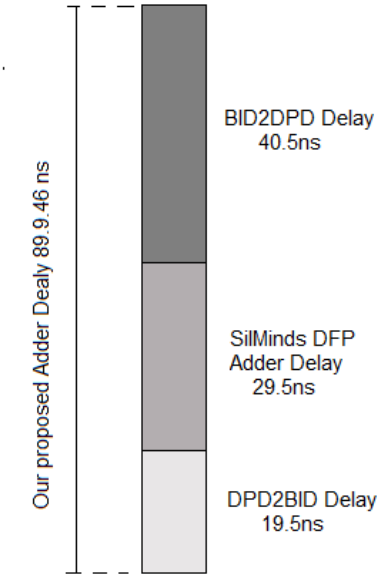
Figure 4.2: The area profiling for our proposed Adder - decimal64



The power consumption of the proposed adder in decimal64 mode is 484 mWatt, this consumed power is divided into 422 mWatt static power and 62 mWatt dynamic power.

The time delay of our proposed adder is divided between the three units as shown in figure 4.3, the BID2DPD converter unit has the largest delay among the other units.

Figure 4.3: Time delay distribution of the proposed adder - decimal64 mode



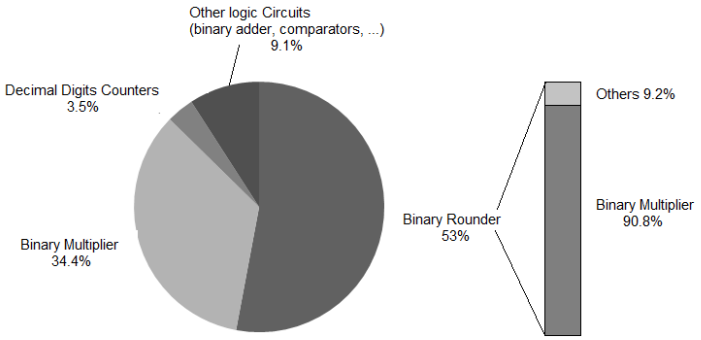
The Schulte Adder is fully implemented on the same FPGA technology Stratix VI. The synthesis results are shown in table 4.3. As shown, the binary rounder has the largest percentage of Schulte area, it has 53% of the total area of the schulte adder design. The binary multiplier 54x64 has large area as well, it covers 34.4% of the total area of Schulte adder design. An important note that the binary rounder module has a multiplier inside it. This multiplier is not the same as the external multiplier 54x64. Its size is 64x65 and it has 90.8% as shown in figure 4.4. It is used to get the rounded result using reciprocal multiplication method. The two used multipliers have an area of 12903 LUTs on Stratix VI FPGA family, this area is 82.5% of the overall adder area (15638 LUTs) which is high ratio.

Table 4.3: Synthesis results of Schulte Adder - decimal64

	Area (LUTs)	Percentage
Binary Rounder	8281	53%
Binary Multiplier 64 x 54	5376	34.4%
Decimal Digits Counters	552	3.5%
Other logics(Binary Adder, Comparators, ...)	1426	9.1%

The Area profiling results for the Schulte adder units for decimal64 mode are shown in figure 4.4.

Figure 4.4: The area profiling of Schulte adder design - decimal64



Binary rounder has a very large area, so the area optimization may lead to better Synthesis results. Table 4.4 shows the synthesis results of our proposed binary rounder and Schulte binary rounder. As shown, we have an advantage in the area but the delay becomes worse. In some applications, our design may be used if we are concerned with area. In other applications where area is not the primary concern, Schulte rounder may be the better choice. The Schulte adder synthesis results is changed when we use our proposed binary rounder. The area and delay results will be balanced between our proposed DFP adder and Schulte DFP adder as shown in table 4.5. The area, Delay and Power consumbtion values are in between our proposed Adder design and Schulte adder design results. The Schulte adder design with our proposed rounder has better area and power consumbtion results comparing to the schulte adder design results without any modifications.

Table 4.4: synthesis results of different binary rounder designs - decimal64

	Area (LUTs)	Delay (ns)	Area Delay Product (KLUT.ns)
The Proposed Rounder design	3188	59	188.092
Schulte Rounder design	8281	34.6	286.522

For decimal128, we implemented the proposed design on Stratix VI FPGA. Schulte Adder is not implemented totally, we implement the largest percentage area units on Stratix VI FPGA, 114x127 binary multiplier and 127x128 binary multiplier and estimate other logic in the adder. This other logic contains Decimal Digits Counters, Binary Adders, Comparators, Rounder control logic and Look Up Tables. The other logic area has 17.5% of the total area in Schulte adder design for decimal64. By analogy, we get the synthesis result as shown in Table 4.6. The implemented binary multipliers synthesis results and the estimated other logic synthesis results are shown in Table 4.7.

Table 4.5: Synthesis results of the different DFP adders - decimal64

	Area (LUTs)	Delay (ns)	Power (mW)	Area Delay Product (KLUT.ns)
The proposed Adder design	6787	89.46	484	607.165
Schulte's Adder design	15638	40.18	647	628.335
Schulte's Adder design with our proposed rounder	10645	64.68	545	688.519

Table 4.6: Synthesis results of the implemented decimal floating point adders (the proposed design and the Schulte Adder design - decimal128)

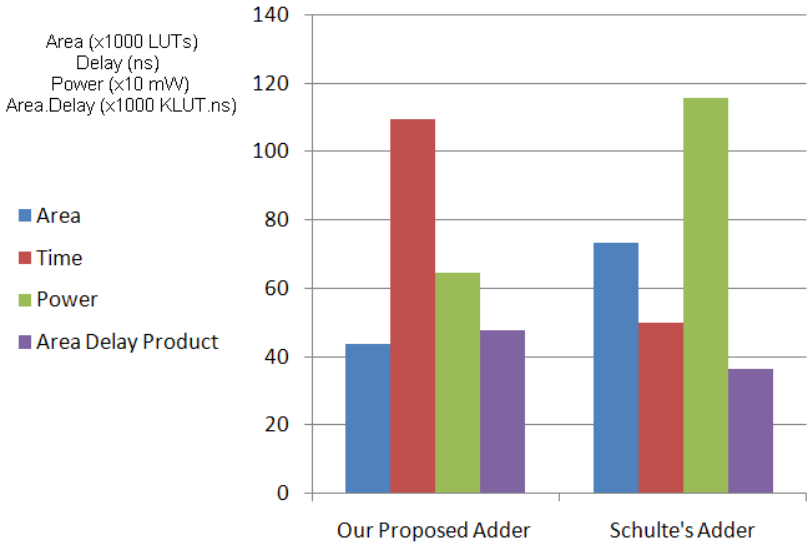
	Area (LUTs)	Delay (ns)	Power (mW)	Area Delay Product (KLUT.ns)
The proposed Adder design	43661	109.5	644	4780.88
Schulte's Adder estimated results	73168	49.9	1155	3651.11

Table 4.7: Synthesis results of the binary multipliers in Schulte adder for decimal128

	Area (LUTs)	Delay (ns)	Area Delay Product (KLUT.ns)
114x127 binary multiplier	25516	41.14	1049.728
127x128 binary multiplier	34848	42.86	1493.585

From table 4.6, Our proposed adder area is less than Schulte adder area in decimal128 mode. Our proposed design delay is more than the double value of Schulte adder design delay. In our design, we have 109.5 ns time delay while in Schulte adder design, the time delay is 49.9 ns. We have a better result in the power consumption, our design has 644 mWatt while 1155 mWatt in Schulte adder design. Figure 4.5 shows the synthesis results for both designs.

Figure 4.5: Comparison between the synthesis results of the proposed Adder and Schulte Adder - decimal128



Our proposed adder design is divided into three main parts, the synthesis results are shown in table 4.8.

Table 4.8: Synthesis results of the proposed Adder - decimal128

	Area (LUTs)	Delay (ns)
BID2DPD Converter A	14910	43.2
BID2DPD Converter B	14910	43.2
SilMinds DFP Adder	5903	41.7
DPD2BID Converter	7938	24.6

The following chart 4.6 shows the area profiling for our proposed Adder in decimal128 mode.

The time delay of our proposed adder is divided between the three units as shown in figure 4.7, the BID2DPD converter unit has the largest delay among the other units.

The power consumption of the proposed adder in decimal128 mode is 644 mWatt, this consumed power is divided into 424 mWatt static power and 220 mWatt dynamic power.

Figure 4.6: The area profiling for our proposed Adder - decimal128

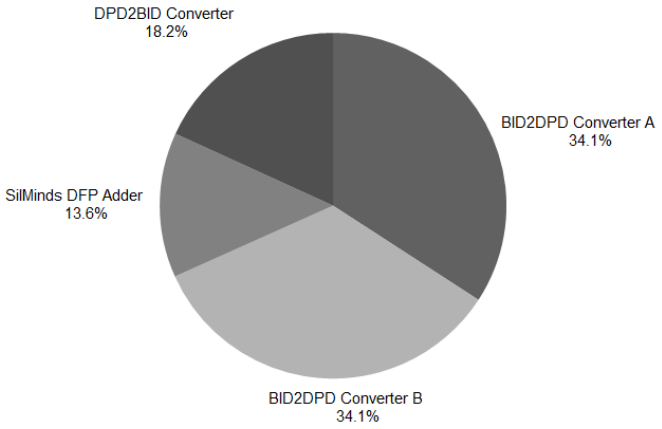
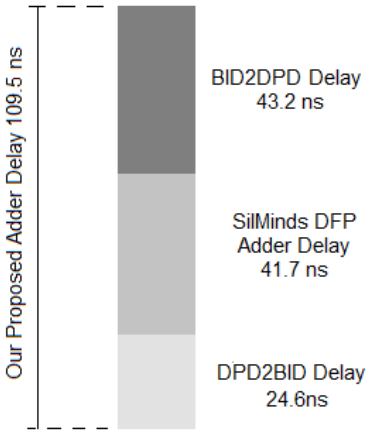


Figure 4.7: The time delay of our proposed adder units - decimal128



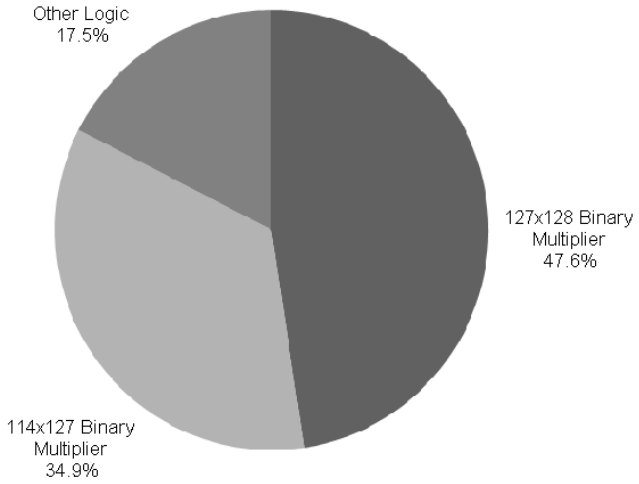
In decimal128, the Schulte Adder synthesis results are estimated, it is not implemented totally on FPGA technology Stratix VI as . The estimated synthesis results are shown in table 4.9. As shown, the 127x128 binary multiplier which is used in the binary rounder to get the rounded result using reciprocal multiplication method, has the largest percentage of Schulte adder area, it has 47.6% of the total area of the schulte adder design. The binary multiplier 114x127 has large area as well, it covers 34.9% of the total area of Schulte adder design. The two used multipliers has area of 60364 LUTs on Stratix VI FPGA family, this area is 82.5% of the overall estimated adder area (73168 LUTs) which is high ratio.

Table 4.9: Synthesis results of Schulte Adder - decimal128

	Area (LUTs)	Percentage
127x128 binary multiplier	34848	47.6%
114x127 binary multiplier	25516	34.9%
Other logics(Decimal Digits Counter, Binary Adder, Comparators, Rounder control logic, ...)	12804	17.5%

The Area profiling results for the Schulte adder units for decimal128 mode are shown in figure 4.8.

Figure 4.8: The area profiling of Schulte adder design - decimal128



Chapter 5: Conclusions and Suggestions For Future Work

5.1 Conclusion

In this thesis, we present a novel architecture for the decimal floating point adder for decimal64 and decimal128 formats.

In decimal64, We achieved 56% reduction in the area, 25% reduction in the power consumption and 4% reduction in the area delay product. compared to Schulte DFP adder design.

In decimal128, we have 24% reduction in area but we have 158% increase in the time delay.

Our design uses SilMinds DFP adder which operates in decimal encoding, We convert from/to binary encoding at the input and output using decimal floating point converters. Schulte DFP adder was implemented for decimal64 and its synthesis results were estimated for decimal128. to compare with our proposed design results.

The two novel Converters were filed at the United States Patents and Trademark Office (USPTO):

- "DPD/BCD to BID converters", October 2012, US Patent application number 13644374.
- "BID to BCD/DPD converters", October 2012, US Patent application number 13657458.

5.2 Future Work

In this section, some ideas are suggested to work on and to extend our work:

- In decimal128 mode, Some block are very large in area as in our proposed design (Binary to BCD converter) and in Schulte design (the binary rounder, the binary multiplier). More optimization are needed for these blocks.
- An ASIC implementation of both designs using 0.11 μm LSI standard cells library is needed. We need to use this technology as the synthesis ASIC results in Shulte design were performed using the LSI Logic Gflxp 0.11 μm CMOS Standard Cell Library [43].
- For other decimal floating point operations as Multiplication, Division, Power, Square root, we can use our proposed converters in the rounder design. The BID rounders can use the same idea in subsection 3.3.1.2.7 which use Binary to BCD

converter then do a simple rounding operation in decimal by shifting decimal digits then converting back the rounded result using BCD to Binary converter.

Our proposed DFP adder can not be used in these units. For example, Divders and Square root units use subtractive or multiplicative methods in thier designs, they need binary or decimal adders/subtractors and/or binary multipliers which deal with integers not floating point numbers.

References

[1] Binary to binary coded decimal converter, Nov. 30 1971. US Patent 3,624,374.

[2] IEEE standard for binary floating-point arithmetic. *ANSI/IEEE Std 754-1985* (1985).

[3] Gcc, the gnu compiler collection. *http://gcc.gnu.org/* (May 2008).

[4] IEEE standard for floating-point arithmetic. *IEEE Std 754-2008* (2008), 1–70.

[5] Sparc64 x : Fujitsus new generation 16 core processor for the next generation unix server. *Hot Chips 25 - California, http://www.fujitsu.com/* (2012).

[6] Aharoni, M., Maharik, R., and Ziv, A. Solving constraints on the intermediate result of decimal floating-point operations. In *Computer Arithmetic, 2007. ARITH'07. 18th IEEE Symposium on* (2007), IEEE, pp. 38–45.

[7] Al-Khaleel, O., Al-Qudah, Z., Al-Khaleel, M., Papachristou, C., and Wolff, F. Fast and compact binary-to-bcd conversion circuits for decimal multiplication. In *Computer Design (ICCD), 2011 IEEE 29th International Conference on* (2011), IEEE, pp. 226–231.

[8] Benedek, M. Developing large binary to bcd conversion structures. *Computers, IEEE Transactions on 100*, 7 (1977), 688–700.

[9] Bhattacharya, J., Gupta, A., and Singh, A. A high performance binary to bcd converter for decimal multiplication. In *VLSI Design Automation and Test (VLSI-DAT), 2010 International Symposium on* (2010), IEEE, pp. 315–318.

[10] Boothroyd, D. C., Eckard, C. B., Lange, R. E., Shelly, W. A., and Yoder, R. W. Binary to binary coded decimal and binary coded decimal to binary conversion in a vlsi central processing unit, Oct. 5 1993. US Patent 5,251,321.

[11] Busaba, F. Y. Method for binary to decimal conversion, Apr. 9 2002. US Patent 6,369,725.

[12] Carlough, S. R., Fleischer, B. M., Li, W. H., and Schwarz, E. M. System and method for performing decimal to binary conversion, Feb. 9 2010. US Patent 7,660,838.

[13] Cinnaminson, C. M. W. Decimal to binary conversion, May 18 1971. US Patent 3,579,267.

[14] Couleur, J. F. Bidec -a binary-to-decimal or decimal-to-binary converter. *Electronic Computers, IRE Transactions on*, 4 (1958), 313–316.

[15] Cowlishaw, M. A summary of densely packed decimal encoding. *IEE ProceedingsComputers and Digital Techniques, ISSN*, 1350–2387.

[16] Cowlishaw, M. Decimal to binary coder/decoder, Aug. 20 2002. US Patent 6,437,715.

[17] Cowlishaw, M. Binary to decimal coder/decoder, Feb. 25 2003. US Patent 6,525,679.

[18] Cowlishaw, M. The decnumber c library. *IBM UK Laboratories, http://www2. hursley. ibm. com/decimal* (2010).

[19] David, F. Binary-decimal converter, Oct. 5 1971. US Patent 3,611,349.

[20] Deschamps, J.-P., Bioul, G. J., and Sutter, G. D. *Synthesis of arithmetic circuits: FPGA, ASIC and embedded systems*. John Wiley & Sons, 2006.

[21] Erle, A, M., Schulte, J, M., Linebarger, and M, J. Potential speedup using decimal floating-point hardware. In *Signals, Systems and Computers, 2002. Conference Record of the Thirty-Sixth Asilomar Conference on* (2002), vol. 2, IEEE, pp. 1073–1077.

[22] Fahmy, H., Raafat, R., Abdel-Majeed, A. M., Samy, R., ElDeeb, T., and Farouk, Y. Energy and delay improvement via decimal floating point units. In *Computer Arithmetic, 2009. ARITH 2009. 19th IEEE Symposium on* (2009), IEEE, pp. 221–224.

[23] Flora, L. P., and Wiener, D. P. Bcd-to-binary converter, May 25 1982. US Patent 4,331,951.

[24] Fowler, R. W. Fractional binary to decimal converter, April 1977. US Patent 4,016,560.

[25] Guild, H. Fast decimal-binary conversion. *Electronics Letters 5*, 18 (1969), 427–428.

[26] Horward, S., Rathbun, M., and Pivovonsky, M. Binary to decimal conversion method and apparatus, Mar. 10 1968. US Patent 3,373,269.

[27] Jaberipur, G., and Kaivani, A. Binary-coded decimal digit multipliers. *Computers & Digital Techniques, IET 1*, 4 (2007), 377–381.

[28] James, R., Shahana, T., Jacob, K., and Sasi, S. Decimal multiplication using compact bcd multiplier. In *Electronic Design, 2008. ICED 2008. International Conference on* (2008), IEEE, pp. 1–6.

[29] Lutz, B. C., and Sather, D. C. Serial binary number and bcd conversion apparatus, Jan. 13 1976. US Patent 3,932,739.

[30] Mathew, S. K., and Krishnamurthy, R. Binary-to-bcd conversion, Jan. 13 2009. US Patent 7,477,171.

[31] Miller, A. J. Binary to binary coded decimal converter, Jan. 17 1978. US Patent 4,069,478.

[32] Nicoud, J.-D. Iterative arrays ror radix conversion. *Computers, IEEE Transactions on 100*, 12 (1971), 1479–1489.

[33] Oracle. BigDecimal (Java 2 Platform SE 5.0). , *http://docs.oracle.com/javase/1.5.0/docs/api/java/math/BigDecimal.html* (2004).

[34] Rhyne, V. T. Serial binary-to-decimal and decimal-to-binary conversion. *Computers, IEEE Transactions on 100*, 9 (1970), 808–812.

[35] Sather, D. C. Bb tsr, Feb. 1 1975. US Patent 3,866,213.

[36] Sayed-Ahmed, A. A., Fahmy, H. A., and Hassan, M. Y. Three engines to solve verification constraints of decimal floating-point operation. In *Signals, Systems and Computers (ASILOMAR), 2010 Conference Record of the Forty Fourth Asilomar Conference on* (2010), IEEE, pp. 1153–1157.

[37] Schmookler, and MS. High-speed binary-to-decimal conversion. *Computers, IEEE Transactions on 100*, 5 (1968), 506–508.

[38] Schwarz, E. M., Kapernick, J. S., and Cowlishaw, M. F. Decimal floating-point support on the ibm system z10 processor. *IBM Journal of Research and Development 53*, 1 (2009), 1–4.

[39] Tang, P. Binary-integer decimal encoding for decimal floating-point. *Intel Corporation, Available at http://754r. ucbtest. org/issues/decimal/bid_rationale. pdf* (2005).

[40] Thompson, J., Karra, I., and Schulte, M. J. A 64-bit decimal floating-point adder. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI* (2004), pp. 297–298.

[41] Took, P. C. Decade circuit, Jan. 2 1972. US Patent 3,638,002.

[42] TOOLE, P. High speed direct binary to binary coded decimal converter and scaler(digital converter for scaling binary number to binary coded decimal number of higher multiple)[patent]. *Patent NumberUS-PATENT-3, 697, 733; US-PATENT-APPL-SN-98772; US-PATENT-CLASS-235-155; US-PATENT-CLASS-340-347 DD* (1972).

[43] Tsen, C., González-Navarro, and Schulte, M. Hardware design of a binary integer decimal-based floating-point adder. In *Computer Design, 2007. ICCD 2007. 25th International Conference on* (2007), IEEE, pp. 288–296.

[44] Tsen, C., Schulte, M., and González-Navarro, S. Hardware design of a binary integer decimal-based ieee p754 rounding unit. In *Application-specific Systems, Architectures and Processors, 2007. ASAP. IEEE International Conf. on* (2007), IEEE, pp. 115–121.

[45] Tyreso, A. Arrangement for converting a binary number into a decimal number in a computer, 1971. US Patent 3,627,998.

[46] Veeramachaneni, S., and Srinivas, M. Novel high-speed architecture for 32-bit binary coded decimal (bcd) multiplier. In *Communications and Information Technologies, 2008. ISCIT 2008. International Symposium on* (2008), IEEE, pp. 543–546.

[47] Wang, L.-K., Schulte, M., Thompson, J., and Jairam, N. Hardware designs for decimal floating-point addition and related operations. *Computers, IEEE Transactions on 58*, 3 (2009), 322–335.

[48] Wang, L.-K., Tsen, C., Schulte, M., and Jhalani, D. Benchmarks and performance analysis of decimal floating-point applications. In *Computer Design (ICCD), 2007 25th International Conference on* (2007), IEEE, pp. 164–170.

[49] Wang, M. C. Binary coded decimal to binary conversion, Aug. 18 1970. US Patent 3,524,976.

[50] Webb, C. IBM z6 the next-generation mainframe microprocessor. *IBM Systems Technology Group, http://speleotrove.com/decimal/IBM-z6-mainframe-microprocessor-Webb.pdf* (2008).

[51] Wiener, D. P. Bcd to binary converter, Apr. 13 1982. US Patent 4,325,056.

[52] Wikipedia. Decimal floating point. *http://en.wikipedia.org/wiki/Decimal_floating_point*.

[53] Yamauchi, S. Method and system for binary-to-decimal interconversion, Jan. 12 1988. US Patent 4,719,450.

ملخص البحث

يعتبر تصميم الدوائر الحسابية للأعداد الكسرية العشرية من الموضوعات البحثية المثيرة للإهتمام في العقد الأول من القرن الواحد والعشرين.

يمكن تمثيل الأعداد الكسرية العشرية بطريقتين: الترميز العشري للكسور العشرية ويشار إليه ب "Densely Packed Decimal (DPD)", الترميز الثنائي للكسور العشرية ويشار إليه ب "Binary Integer Decimal (BID)". تستخدم أغلب هذه الدوائر الترميز العشري للكسور العشرية إلى الآن لأن التعامل مع التصميمات التي تعمل بالترميز الثنائي للكسور العشرية غير فعال.

في هذه الرسالة, تم اقتراح تصميم دوائر جديدة للمحولات من الترميز الثنائي للكسور العشرية الى الترميز العشري للكسور العشرية والعكس, وإحاطة هذه المحولات لدوائر تقليدية تعمل بالترميز العشري للكسور العشرية.

تم إثبات أن هذا التحويل من الترميز الثنائي إلى الترميز العشري للكسور العشرية يليه وحدة جامع الأعداد ذات الترميز العشري للكسور العشرية يليه تحويل مرة أخرى من الترميز العشري إلى الترميز الثنائي للكسور العشرية, أفضل من إستخدام الجامع للأعداد ذات الترميز الثنائي للكسور العشرية بطريقة مباشرة.

بمقارنة نتائج الجامع الذي يستخدم الطريقة المقترحة في هذه الرسالة, ونتائج منشورة للتصميم الذي يعمل بالترميز الثنائي للكسور العشرية, نجد التالي: انخفاض في المساحة بمقدار ٥٦.٦% وانخفاض في استهلاك الطاقة بمقدار ٢٥% وانخفاض في مضروب المساحة وإستهلاك الوقت بمقدار ٤%.

دائرة جمع للكسور العشرية الممثلة بالترميز الثنائي

اعداد
أحمد عبدالمرضي محمد أيوب

رسالة مقدمة إلى كلية الهندسة – جامعة القاهرة
كجزء من متطلبات الحصول على درجة الماجستير
في
هندسة الإلكترونيات والإتصالات الكهربية

يعتمد من لجنة الممتحنين:

_____

أ.د. : أمين محمد نصار المشرف الرئيسى

_____

د. : حسام علي حسن فهمي عضو

_____

د. : إبراهيم قمر الممتحن الداخلي

_____

أ.د. : السيد مصطفى سعد الممتحن الخارجي
الأستاذ بكلية الهندسة – جامعة حلوان

كليــة الهندســة - جامعــة القاهــرة
الجيـزة - جمهوريـة مصـرالعربيــة
٢٠١٤

دائرة جمع للكسور العشرية الممثلة بالترميز الثنائي

اعداد
أحمد عبدالمرضي محمد أيوب

رسالة مقدمة إلى كلية الهندسة – جامعة القاهرة
كجزء من متطلبات الحصول على درجة الماجستير
في
هندسة الإلكترونيات والإتصالات الكهربية

تحت اشراف

أ.د. أمين محمد نصار
أستاذ , كلية الهندسة , جامعة القاهرة

د. حسام علي حسن فهمي
أستاذ مساعد, كلية الهندسة, جامعة القاهرة

كليــة الهندســة - جامعــة القاهــرة
الجيـزة - جمهوريـة مصـر العربيــة

٢٠١٤

دائرة جمع للكسور العشرية الممثلة بالترميز الثنائي

اعداد

أحمد عبدالمرضي محمد أيوب

رسالة مقدمة إلى كلية الهندسة – جامعة القاهرة
كجزء من متطلبات الحصول على درجة الماجستير
في
هندسة الإلكترونيات والإتصالات الكهربية

كليــة الهندســة - جامعــة القاهــرة
الجيـزة - جمهوريـة مصـر العربيــة

٢٠١٤