IEEE 754-2008 COMPLIANT DECIMAL FLOATING POINT DIVIDER

By

Ahmed Hamdy Ahmed Khalil

A Thesis Submitted to the
Faculty of Engineering at 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
2015

IEEE 754-2008 COMPLIANT DECIMAL FLOATING POINT DIVIDER

By

Ahmed Hamdy Ahmed Khalil

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
Master of Science
in
Electronics and Communications Engineering

Thesis Advisor
Prof. Dr. Hossam A. H. Fahmy

Prof. of Computer Arithmetic, Faculty of Engineering, Cairo University

Faculty of Engineering, Cairo University

Giza, Egypt

2015

IEEE 754-2008 COMPLIANT DECIMAL FLOATING POINT DIVIDER

By

Ahmed Hamdy Ahmed Khalil

A Thesis Submitted to the
Faculty of Engineering at 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. Hossam A. H. Fahmy, Thesis Main Advisor

_____

Associate Prof. Dr. Ahmed Nader Mohieldin, Member

_____

Prof. Dr. El-Sayed Mostafa Saad, Member

(Prof. of Electronic circuits, Faculty of Engineering, Helwan University)

Faculty of Engineering, Cairo University

Giza, Egypt

2015

Acknowledgements

First of all I must thank ALLAH for his great mercy supporting me all the way till the end. If it weren't for his help, I wouldn't have reached this point.

I would like to thank my advisor, Prof. Hossam A. H. Fahmy for giving me the opportunity to work in a fruitful research environment and for his continuous guidance and support, as well as for his successful discussion and encouragement.

Most importantly, I would like to thank my family for their continuous encouragement, and for helping me all through my work.

Contents

List of Figures

List of Tables

List of Abbreviations

AQA	Accurate Quotient Approximation
BFP	Binary floating point
BCD	Binary coded decimal
BID	Binary integer decimal
CSA	Carry save adder
CPA	Carry propagation adder
DFP	Decimal floating point
DPD	Densely packed decimal
FO4	Fan out of four
MSD	Most significant digit
MSB	Most significant bit
MUX	Multiplexer
NaN	Not a Number
LZV	Leading zeros value
PP	Partial product
qNaN	Quite Not a Number
sNaN	Signal Not a Number
SRT	Sweeney, Robertson, and Tocher division algorithm
ZF	Zero flag

Abstract

The binary numbering system is the most suitable for electronic computers, and the decimal numbering system is the most used in human life especially in commercial applications. But the binary system can not represent decimal fraction numbers accurately, so the IEEE 754-2008 standard defines the decimal floating numbers and decimal floating arithmetic operations. The decimal floating point division is one of these operations.

This thesis presents new design and implementation for decimal floating point divider based on Accurate Quotient Approximation algorithm which can be considered as very high radix decimal divider. Accurate Quotient Approximations is an iterative division algorithm based on look-up table that gives an approximation to the reciprocal of the divisor. The algorithm iterates by using the reciprocal to find an approximated value for the quotient. Then the approximated quotient is multiplied by the divisor and the result is subtracted from the partial remainder to find the new partial remainder. At first iteration the partial remainder is the dividend. The divider iterates until the quotient reaches the desired accuracy. This divider can produce three digits per iteration. The operation of this divider is compliant to IEEE 754-2008 standard. Also it supports the five rounding modes that are defined in the IEEE 754-2008 standard, and another two famous rounding methods. This divider is extended to execute decimal floating point division operations.

The functionality of the proposed decimal floating point divider is verified by around one million test cases that are designed to test decimal floating point units. Also the design is synthesized by using Synopsys design compiler tool working on 65nm technology which shows that the critical path delay is 48.2 FO4 and area equals 156842 NAND2 gate.

The main goal of this thesis is to design and implement new decimal floating point divider functionally correct. The proposed DFP divider can produce three quotient digits in each iteration, which can reduce the overall latency.

Chapter 1. Introduction

The decimal number system is the most widely used system for human calculations. So that at the beginning of computer machines age all machines were based on decimal system. Mechanical computers mimicked human numeration systems for scientific and commercial calculations, and the most used system was decimal system. At the start of electronic computer industry many computers used decimal system in arithmetic operations such as ENIAC [1] and IBM 650 [2].

However the nature of memory units and flip-flops is binary and it is required four binary bits to represent each decimal digit which consumes more memory than binary system. Furthermore the binary system arithmetic units are simpler and faster than decimal system arithmetic units. Therefore Burks, Goldstine and von Neumann [3] discussed the superiority of binary system over decimal system. They announced that using binary system for addressing, data storage and arithmetic operations is more suitable for electronic computers due to the nature of these computers which have two statuses for the signal. Therefore the binary system dominates most of computers nowadays and there are few computers based on decimal system.

The rest of this chapter is organized as follows: Section1.1 explains the increasing importance of decimal arithmetic. Section1.2 discusses the decimal floating point standard format with its arithmetic operations. And finally section 1.3 surveys the recently published hardware implementations for different decimal floating point operations.

1.1 The Need for Decimal Floating Point Arithmetic

Although binary based computers dominate the world, decimal computations can't be ignored. Decimal numeration system is essential for many applications [4]. Databases belong to 51 commercial and financial organizations were surveyed and investigated, these databases include many financial applications such as banking, billing, inventory control, financial analysis, taxes, and retail sales. There were more than 456,420 columns which contained numeric data and were investigated to extract statistic information. This survey reported that 55% were decimal, and that further 43.7% were integer types which could have been stored as decimal numbers [4]. The results of these applications are required to be accurate and rounded correctly to be committed by human manual calculations and law.

For binary based computers, decimal numbers will be converted to/from binary numbers. Decimal numbers may not be converted exactly, due to the lack of binary system accuracy and finite precision hardware. Most of fraction numbers are not converted to binary numbers properly [5], let the decimal number $X_{dec} = 0.6$, to convert this decimal number to binary number it will be $X_{bin} = 0.10011001100110011001100110 \ldots \ldots$ that requires infinite number of bits to be represented exactly in binary which is not available so this number will be approximated, the stored value will be $X_{dec}^{stored} = 0.599999999999999999913263$, so any operation using this number will produce inaccurate results although the arithmetic operation is correct. The decimal to/from binary conversion is implemented using software programs with high delays.

In addition to the accuracy problem there is another problem caused by binary arithmetic is the removal of trailing fraction zeroes. For example, binary system can't distinguish between

1.5 and 1.50 because of the normalization nature of binary system. The trailing fraction zeros are essential in the calculation, they are very important for physics measurement for example if it is reported that, the mass of a body is 10.7 kg versus 10.700 kg the two measures are not the same as the first one is accurate for 0.1 kg but the second one is accurate for 0.001 kg. Hence binary arithmetic units can't be used directly for financial application and decimal arithmetic operations as they produce results not compatible with law and human requirements [5].

To avoid the drawbacks of using binary arithmetic units for decimal calculations, researchers and companies made a lot of efforts to overcome these drawbacks. Finally we can find two methods to deal with decimal calculations the first, is to keep decimal numbers without conversion and software libraries will execute the decimal calculations using binary floating point (BFP) arithmetic units. The most widely used libraries are Sun's BigDecimal for Java [6], IBM's decNumber library [7], and Intel's Decimal Floating-Point Math library [8]. But the performance of software libraries is very bad, as the performance can reach 1000 time slower than implementing dedicated hardware [9].

The other method is to implement pure hardware for decimal operations to overcome the software bad performance, IBM presented Z900 as one of the first microprocessor that contains decimal integer arithmetic unit [10], although this unit is limited on decimal integers, decimal fixed point operations can be executed by scaling the operand using software libraries.

Fixed point and integer arithmetic units can generate accurate results, but the scaling of floating-point numbers, and rounding of the final result limit their usage. The width of these units is limited by the format precision, so when using fixed point and integer arithmetic units for floating point operations the input operands will be scaled to convert them to integer numbers this scaling operation is implemented using software programs, this process is difficult to be executed and can result in errors specially when using very large values and very small values for the same operation [4].

Although rounding is very important for financial and commercial applications, fixed point and integer arithmetic units don't implement round process in hardware but it is required to use software to do that. Using software for scaling and rounding processes impose high delay on the operation, also consumes more power than using dedicated hardware for all operation [4].

Therefore it is required to implement pure hard ware for decimal floating point arithmetic to avoid drawbacks of binary floating point arithmetic units and to save time and power of software used to adjust the operation of fixed point and integer arithmetic units.

1.2 IEEE 754-2008 Standard

IEEE 754-1985 is the most widely used standard for binary floating point arithmetic, and is implemented for many microprocessor designs and software libraries [11]. But this is a binary floating-point standard so it cannot be used for decimal arithmetic. Then IEEE published floating point standard IEEE 854-1987 [12], which was radix independent, and this standard was designed mainly for scientific and engineering applications, so this standard didn't find a way for commercial needs. Due to booming of decimal floating point arithmetic

and there was not any standard to identify decimal floating point (DFP) arithmetic, IEEE published the IEEE 754-2008 in August 2008 [13].

IEEE 754-2008 standard defines decimal floating-point data formats and operations as follow,

1.2.1 DFP Formats and Encoding

The standard specifies three formats for floating point decimal numbers, two are basic formats (decimal64 and decimal128) and one is storage format (decimal32), as shown in Table 1.1. The basic formats are used for DFP arithmetic operations, and the storage format is not used by DFP arithmetic operations. These formats can represent positive and negative values within the range of the used precision format, $\pm 0, \pm \infty$, and NaN (Not a Number).

Decimal format	Decimal 32	Decimal 64	Decimal 128
Total storage width	32	64	128
Combinational field	11	13	17
Trailing significand field	20	50	110
Total significand digits	7	16	34
Maximum Exponent	96	384	6144
Minimum Exponent	-95	-383	-6143
Exponent Bias	101	398	6176
Exponent width	8	10	14

Table 1.1: Decimal formats defined in IEEE 754-2008

The representation of decimal floating-point number is:

$$(-1)^S \times \mathrm{C} \times 10^{\mathrm{q}} \tag{1.1}$$

Where S is the sign, q is the exponent, $\mathrm{C} = (\mathrm{d}_{\mathrm{p}-1}\mathrm{d}_{\mathrm{p}-2} \ldots\ldots \mathrm{d}_0)$ is the significand, or coefficient, where $\mathrm{d}_\mathrm{i} \in \{0,1,2,3,4,5,6,7,8,9\}$, and p is the precision. There are two restrictions on C and q [13]:

- C must be an unsigned integer between 0 and $(10^{\mathrm{p}} - 1)$. Notice that in the standard, C is not required to be normalized. So we can find different representation for the same number, let 500×10^3 and 5×10^5 have same value, and both belong to the same cohort.
- q must be an integer satisfying equation, $1 - \mathrm{E}_{\min} \leq \mathrm{q} + \mathrm{p} - 1 \leq \mathrm{E}_{\max}$.

Figure 1.1 shows the decimal interchange floating-point format adopted in the IEEE 754-2008 standard which includes,

S Sign Field	G Combination Field	T Trailing Significand Field

Figure 1.1: Decimal interchange floating-point format

Sign bit S represents the sign of the decimal number, it has two statuses one or zero then the decimal number will be negative or positive respectively.

The second part is the combination field G; it can be divided into two subfields. The first subfield consists of the most significant five bits of G, is used to encode the MSD of the significand, the most significant two bits of the exponent, and is used to indicate NaN and infinity cases as shown in table 1.2. The second subfield consists of the remaining bits of the biased exponent, as the exponent is encoded in binary excess-code and the bias value differs according to the precision format as shown in table 1.1.

The remaining part is the trailing significand field T this field consists of the remaining fifteen digits of the significand. For trailing significand field value there are two types of encoding formats defined in the standard. Densely Packed Decimal "DPD" is introduced by IBM. DPD format encodes every three decimal digits using 10 bits [14]. DPD format is efficient in memory storage and data buses usage than Binary Coded Decimal "BCD" which uses 4 bits for each single digit. But it is not easy to use DPD directly for arithmetic operations, so there are need to convert from DPD format to/from BCD format [5]. Conversion process can be implemented using dedicated circuits which can cost low delay. Intel presented Binary Integer Decimal "BID"; here the entire trailing significand field value is converted from decimal form to binary form to be stored [15]. The BID encoding format is suitable for software implementation, to use the binary integer arithmetic logic unit in the current microprocessors [5].

Special values are encoded as follow:

- If the most significant five bits of the combination filed are 11110, then we have $\pm\infty$, according to the sign bit.
- If the most significant five bits of the combination filed are 11111, then we have NaN case, and if the most significant sixth bit is 1 hence we have sNaN otherwise we have qNaN, and all the other bits are ignored. sNaN represents values for uninitialized variables or missing data samples. qNaN results from any invalid operations or operations that involve qNaN as operand.
- Overflow occurs when the absolute value of the result is greater than the maximum available value$(10^{p} - 1) \times 10^{E_{max}-p+1}$.
- Underflow occurs when the result absolute value is smaller than $10^{E_{min}-p+1}$ and not equals the zero.
- Subnormal number has a value between $10^{E_{min}}$ and $10^{E_{min}-p+1}$. Subnormals fill the underflow gap around zero in DFP arithmetic.

G's MSD(5-bits)	type	Exponent's MSB (2-bits)	Significand's MSD
$a_i a_{i-1} a_{i-2} a_{i-3} a_{i-4}$	$D_0 \leq 7$	$a_i a_{i-1}$	$0 a_{i-2} a_{i-3} a_{i-4}$
$11 a_{i-2} a_{i-3} a_{i-4}$	$D_0 > 7$	$a_{i-2} a_{i-3}$	$100 a_{i-4}$
$11111 - \begin{matrix}0\\1\end{matrix}$	qNaN sNaN	-	-
11110	Infinity	-	-

Table 1.2: Decoding combination field

1.2.2 DFP Operation

To implement DFP hardware compliant with IEEE 754-2008 standard, this hardware must support at least one of the standard basic formats (decimal64 or decimal128). The operands of DFP operations arrive with a compact format (DPD or BID), then the significand, biased exponent, sign, and the flags for each operand are extracted. All operations are executed to generate intermediate results as if having an infinite precision. Then this intermediate result is rounded according to the desired round mode and the precision of the destination significand, and then all final result components (significand, biased exponent, sign, and flags) are combined back to the desired DFP format.

The IEEE 754-2008 standard DFP operations are classified mainly as follow:

- Basic operations, which are addition, subtraction, multiplication, division, square-root, and fused multiply addition. This includes correct rounding after each operation for inexact results according to standard rounding recommendations.
- Two new DFP arithmetic operations, which are Samequantum and Quantize. As $\mathrm{Samequantum}(A, B)$ operation is used to compare the exponent of A and B, the output is true if both are the same or false otherwise. And $\mathrm{Quantize}(A, B)$ operation produces a DFP number with the significand of operand A and the exponent of operand B.
- DFP comparison operations, here the numerical value of two DFP numbers are compared in-respect the cohort of these numbers.
- Conversion of DFP numbers, this function can convert among DFP, BFP and integer formats. Conversion between DFP and BFP must be rounded correctly.
- DFP elementary operations, the standard defines recommendations to provide exact rounding for the results of elementary operations, such as logarithms, and exponentials.

The DFP numbers are non-normalized, so one DFP number can have multiple representations; all these representations are called DFP number's cohort. The standard defines the term preferred exponent which is the suitable exponent for the result. If the result is inexact the preferred exponent will be the smallest available exponent to keep the maximum number of significand digits. But if the result is exact the preferred exponent will differs according to the type of the arithmetic operation. For example the preferred exponent for the division operation is the result of subtracting the divisor's exponent is from the dividend's exponent.

1.2.3 DFP Rounding Modes

Rounding process is very important after every DFP operation, as all operations produce an intermediate result with infinite precision, so it is required to round this result to finite precision to be suitable for the destination precision format. IEEE 754-2008 standard defines five rounding modes for DFP arithmetic operations as follow,

- **roundTiesToEven**: the absolute result is rounded to nearest number. If tie case occurs the absolute result is rounded to nearest even value.

- **roundTiesToAway**: the absolute result is rounded to nearest number. If tie case occurs the absolute result is rounded to the larger number.
- **roundTiesToPositive**: the result is rounded towards positive infinity (if the final result sign is positive then the result is rounded up, else the extra digits are truncated).
- **roundTiesToNegative**: the result is rounded towards negative infinity (if the final result sign is negative then the absolute result is rounded away from zero, else the extra digits are truncated).
- **roundTowardZero**: the absolute result is rounded towards zero, (all extra digits are truncated).

1.2.4 Exception Handling

Exception arises when the result of the DFP operation is not an expected DFP number, so the corresponding exception flag is signaled and the default exception format is delivered. The IEEE 754-2008 standard defines five exceptions as follows,

- Invalid operation: is signaled when there is invalid arithmetic operation such as, $\infty \times 0$ multiplication, $\frac{0}{0}$ division operation, and square-root of negative operands or there are NAN operands. The default result is qNaN.
- Division by zero: when the dividend of a DFP division operation is a finite non-zero number and the divisor is zero, the Division by zero exception is signaled. This operation is a valid operation and the default result is $\pm\infty$ according to the result sign.
- Overflow: when the result value is greater than the maximum available value for the operation format, hence the overflow and inexact exceptions are signaled. There are two default results according to the round mode and the sign of the result. If the absolute result is rounded up the default result will be signed infinity, else the default result will be the maximum available number for destination format $(10^{p} - 1) \times 10^{E_{max}-p+1}$.
- Underflow: when the result value is smaller than $10^{E_{min}-p+1}$ and not equals the zero, the underflow exception is signaled. The default result may be signed zero, a subnormal number with an absolute value lower than $10^{E_{min}-p+1}$, or minimum representable number $\pm 10^{E_{min}-p+1}$ according to the round mode. The inexact exception is signaled if the result is not exact.
- Inexact: the inexact exception is signaled, when the rounded result is not the same like the infinite precision result. The default result is the rounded result.

1.3 Standard Compliant Hardware Implementations

There are many hardware designs have been introduced in the last decade to implement the DFP different operations that specified in the IEEE 754-2008 standard, such as addition/subtractaction, multiplication, division, and some elementary operations.

In the last few years many DFP adders compliant with the standard were produced. Thompson et al. [16] introduced the first DFP adder compliant with the standard, the operation of this adder is enhanced and extended in [17, 18]. Vazquez and Antelo introduced further improvements in [19]. Two different designs, one for high speed and the other for low area, have been introduced in [20].

Multiplication operation is one of the most frequently used operations in DFP arithmetic, so many designs have been proposed. The first standard compliant multipliers can be found in [21, 22]. In [23] Vazquez proposed two DFP multipliers, one is to decrease the area and the other is to enhance the delay.

The division operation didn't have the same attention paid for addition and multiplication processes. However, IBM introduced the first compliant with standard divider in POWER6 [24]. Also Vazquez introduced another divider compliant with the standard in [25].

IBM produced POWER6 microprocessor [26, 27], which is the first micro-processor that implements DFP arithmetic operations compliant with the standard in hardware. It supports both the decimal64 and the decimal128 formats.

1.4 Organization of The thesis

The remaining of this thesis is organized as follows. Chapter 2 provides a detailed survey of the previous studies, Chapter 3 presents the main decimal units that are used to implement the divider. The proposed division algorithm and the detailed design of the proposed divider are provided in Chapter 4. Design verification, implementation, and comparison with others can be found in Chapter 5. Chapter 6 presents the conclusion and suggestions for the future work.

Chapter 2. Previous Work

In order to achieve high performance in executing decimal floating operations, most of researchers tend to perform the four basic decimal operations which are addition, subtraction, multiplication, and division in hardware. Division operation is not as fast as the other three basic operations; the reason is due to the nature of division operation. Since the division is an iterative algorithm, and requires complex algorithm to be performed, so division operation has large latency with respect to the other three basic arithmetic operations.

Division is a very important operation for many applications, such as graphics, digital signal processing, and scientific operations. But researchers have not paid adequate attention to design division units, because division operation is rated as infrequent operation. However inefficient implementation of divider units will degrade the performance of many applications. Simulations have been carried out to show the effect of division and square-root operations on the performance of superscalar processors [28]. This study found that, changes in the density of the division and square root operations below 1% lead to changes in the processor performance round 20%.

Oberman also studies the effect of FP division on overall system performance [29], he found that, when the floating point division operation represents 3% of all floating point operations, this can degrade the system performance up to 40%. Oberman reported that FP adders and FP multipliers consume around 27% and around 18% of the FP dividers result. This shows that inefficient implementation of FP dividers will affect the consumers of the divider results; hence any enhancement to adders and multipliers will be meaningless without high performance dividers.

Incorrect implementation of divider unit can lead to massive financial losses, in 1994 Intel lost US$475 million, due to an error in the division part of the Pentium microprocessor's floating-point unit [30, 31].

The following section presents three types of binary division algorithms. Section 2 introduces two types of decimal division.

2.1 Binary Division Algorithms

Despite the radix difference between decimal system and the binary system, all the decimal division algorithms are derived from binary division algorithms. So in the following we will introduce the main ideas of binary division method.

Division is an iterative algorithm, defined by,

$$X = Q \times Y + R \tag{2.1}$$

Where X is the dividend, Y is the divisor, Q is the quotient, and R is the final remainder.

When maximum relative error of the quotient is 1ulp (unit at last position), which define the accuracy of the divider,

$$|R| \leq |Y| \times ulp \tag{2.2}$$

Division algorithms can be divided into five classes according to [32], which are "digit recurrence, functional iteration, very high radix, lookup table, and variable latency".

Here we will introduce brief description for digit recurrence, functional iteration, and very high radix binary division algorithms.

2.1.1 Digit Recurrence Algorithm

Digit Recurrence algorithms are similar to traditional paper-pencil division method. It is an iterative algorithm that retires a radix-r quotient digit q_i every iteration, and the most significant digit is retired first [33]. The fundamental equation of digit recurrence algorithms is defined as,

$$R_i = r \times R_{i-1} - q_i \times Y \tag{2.3}$$

The process of selection quotient digits q_i is called Quotient Digit Selection,

$$q_i = QDS(r \times R_i ; Y) \tag{2.4}$$

Quotient digits are selected such that,

$$q_i \times Y \leq r \times R_{i-1} < (q_i + 1) \times Y \tag{2.5}$$

And the convergence condition is,

$$-Y \leq R_0 < Y, \quad \text{where} \quad R_0 = X \tag{2.6}$$

After n iterations, when division finishes, the final n-digit quotient is:

$$Q = \sum_{i=0}^{i=n-1} q_i \times r^{-i} \tag{2.7}$$

And the final remainder is calculated as follow,

$$\text{Remainder} = \begin{cases} R_n \times r^{-n} & \text{if } R_i \geq 0 \\ (R_n + Y) \times r^{-n} & \text{if } R_i < 0 \end{cases} \tag{2.8}$$

As shown in the previous equation if R_n is negative a correction step is required by adding the divisor to the negative remainder, and one ulp will be subtracted from the quotient to correct it.

There are many methods to perform digit recurrence division algorithm as follow.

2.1.1.1 Restoring Division

The name "Restoring" comes up because the partial remainder, after each iteration must be kept non negative so that a restore step to the previous values for the partial remainder and the quotient digit q_i is required at the end of the iteration [33]. Here the quotient digits are chosen from $q_i \in \{0,1,2,\dots\dots\dots,r-1\}$ which is a nonnegative digit set, hence the convergence condition will be,

$$0 \leq R_i < Y \tag{2.9}$$

And the quotient digit is selected as follow,

$$q_i = k \quad \text{such that} \quad k \times Y \leq r \times R_{i-1} < (k+1) \times Y \tag{2.10}$$

For binary-radix r = 2 the QDS is defined as,

$$q_i = \begin{cases} 0 & \text{if} \quad 2 \times R_i < Y \\ 1 & \text{if} \quad d \leq 2 \times R_i < 2Y \end{cases} \tag{2.11}$$

2.1.1.2 Non-Restoring Division

For non-restoring division algorithm it is allowed that the remainder to be negative value and the determined quotient digits may be incorrect. The incorrect quotient digit will be corrected by the determined quotient digit in the next iterations [33]. Quotient digits are selected as follow,

$$q_i = \begin{cases} \bar{1} & \text{if} \quad 2 \times R_i < 0 \\ 1 & \text{if} \quad 2 \times R_i \geq 0 \end{cases} \tag{2.12}$$

Non-restoring division algorithm is faster than restoring division algorithm because the partial remainder is compared with zero instead of the divisor; hence the convergence will be,

$$0 \leq R_i \tag{2.13}$$

2.1.1.3 Binary SRT Method

SRT [33] is an advanced way of recurrence binary division as the quotient digit is selected from a redundant digit set,

$$q_i = \begin{cases} \bar{1} & \text{if} \quad 2 \times R_i \leq -Y \\ 0 & \text{if} \quad -Y \leq 2 \times R_i < Y \\ 1 & \text{if} \quad 2 \times R_i < Y \end{cases} \tag{2.14}$$

Which speed up the division process, as in the non-restoring division algorithm it is required to perform addition or subtraction, but is SRT this can be avoided in many iterations due to the quotient digit set includes the zero.

The calculation of remainder still follows equation (2.7).

To avoid comparison of the partial remainder with the divisor, the divisor is normalized to be $\frac{1}{2} \leq |Y| \leq 1$ and the partial remainder will be compared with $\frac{1}{2}$ and $-\frac{1}{2}$ and, the selection of q_i will be,

$$q_i = \begin{cases} \bar{1} & \text{if} \quad 2 \times R_i \leq -0.5 \\ 0 & \text{if} \quad -0.5 \leq 2 \times R_i < 0.5 \\ 1 & \text{if} \quad 2 \times R_i < 0.5 \end{cases} \tag{2.15}$$

2.1.1.4 High Radix SRT Algorithm

As shown in the previously introduced digit recurrence algorithms there is only one bit of the quotient is retired every iteration which leads to increase number of iteration of the division operation, so by changing the radix of the SRT division algorithm to be $r = 2^m$ instead of $r = 2$ the number of iterations will be decreased to be $\left\lceil \frac{n}{m} \right\rceil$ where n is the precision of the desired final quotient [34].

Robertson introduced the first high radix SRT division [35]. In this algorithm, the quotient digits are selected from redundant signed digit set as $q_i \in \{-a, a\}$, where,

$$\left\lceil \frac{r}{2} \right\rceil \leq a \leq r - 1 \tag{2.16}$$

So that convergence equation that applied for high-radix SRT division is rewritten as

$$-\rho Y \leq R_i < \rho Y \tag{2.17}$$

Where ρ is the redundancy factor, and $0.5 \leq \rho = \frac{a}{r-1} \leq 1$

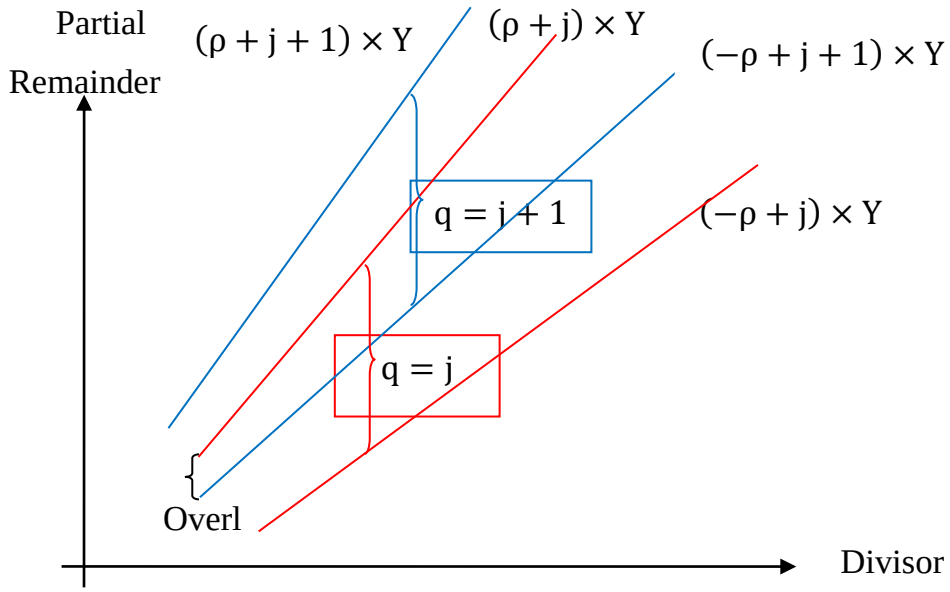

Figure 2.1: Redundant numbering system effect on quotient digit selection

Due to using redundant numbering system, there will be overlap regions, which means more than one quotient digit may be selected. Larger a leads to increase the restricted range of the partial remainder as shown in figure 2.1. But larger a will lead to complex QDS function as quotient digits candidates number is increased. So it is not easy to determine the best redundancy level to be selected as it depends on the algorithms used for QDS function.

2.1.2 Functional Division Algorithms

Functional iteration division algorithms based on functional solution methods to find the quotient, there are two main techniques used, which are Newton-Raphson [32, 36] and series expansion [37, 38]. The following two subsections introduce brief description for both techniques.

2.1.2.1 Newton-Raphson

Instead of determining the quotient directly, Newton-Raphson algorithm calculates an approximation to the reciprocal of the divisor, and then the resulted reciprocal is multiplied by the dividend to produce an approximation of the quotient. The division equation is defined as follow,

$$Q = \frac{X}{Y} \tag{2.18}$$

Then equation (2.17) can be written as,

$$Q = \frac{1}{Y} \times X \tag{2.19}$$

To find the reciprocal using Newton-Raphson algorithm, the following function is defined,

$$f(y) = \frac{1}{y} - Y \tag{2.20}$$

First order Newton-Raphson equation is defined as,

$$y_{i+1} = y_i - \frac{f(y_i)}{f(\acute{y}_i)} \tag{2.21}$$

Newton-Raphson equation (2.21) is applied on equation (2.20), in an iterative manner to calculate an approximation of the reciprocal as follow,

$$y_{i+1} = y_i - \frac{f(y_i)}{f(\acute{y}_i)} = y_i - \frac{\frac{1}{y_i} - Y}{-\frac{1}{{y_i}^2}} = y_i(2 - Y \times y_i) \tag{2.22}$$

Where, $i = 0,1,2, \ldots \ldots \ldots, n$

Here the approximated reciprocal is refined quadratically, because the error is decreased with the same rate,

$$\varepsilon_{i+1} = \varepsilon_i^2 \tag{2.23}$$

After n iterations the reciprocal y_n reaches to the desired precision, then the determined reciprocal is multiplied by the dividend to obtain an approximation for the quotient.

2.1.2.2 Series Expansion

Series expansion is a functional iterative division algorithm, which tries to find the quotient directly using Maclaurin series such that,

$$Q = \frac{X}{Y} = X \times g(y) \tag{2.24}$$

Where the reciprocal of the divisor is defined as,

$$g(y) = \frac{1}{Y} = \frac{1}{1 + y} \tag{2.25}$$

The function $g(y) = \frac{1}{1+y}$ is expanded using Maclaurin series to be,

$$g(y) = \frac{1}{1 + y} = 1 - y + y^2 - y^3 + y^2 - \cdots \tag{2.26}$$

The divisor is normalized such that $0.5 \leq Y < 1$, hence $|y| \leq 0.5$, so above equation can be written as,

$$g(y) = (1 - y)(1 + y^2)(1 + y^4)(1 + y^8)(1 + y^{16})(1 + y^{32}) \ldots \tag{2.27}$$

Hence the quotient equation will be written as,

$$Q = X \times [(1 - y)(1 + y^2)(1 + y^4)(1 + y^8)(1 + y^{16})(1 + y^{32}) \ldots] \tag{2.28}$$

The above equation requires infinity number of factors to produce the exact quotient which can't be achieved, so an approximated version of the quotient is calculated in an iterative manner to certain accuracy as follow,

Let the approximated quotient at iteration i is,

$$Q_i = \frac{X_i}{Y_i} \tag{2.29}$$

Where X_i and Y_i are scaled value of the dividend and the divisor at iteration i respectively, such that $\frac{1}{Y_i}$ tends to value 1, hence X_i will move towards the quotient. Let $X_0 = X$ and $Y_0 = Y$. X_i and Y_i are calculated as follow,

$$X_i = R_{i-1} \times X_{i-1} \quad \text{and} \quad Y_i = R_{i-1} \times Y_{i-1} \tag{2.30}$$

Where $R_i = 2 - Y_i$, is the scaling factor and equals the two's complement of the scaled divisor at iteration i.

This algorithm convergence quadratically, hence after n iterations the error will be,

$$\varepsilon_n = Q - X_n \leq 2^{2n} \tag{2.31}$$

2.1.3 Very High Radix Division Algorithm

Very High radix division algorithm is an algorithm where large numbers of bits are retired per iteration. Wong and Flynn [39], and Matula [40] independently proposed the very high

radix division algorithm. Both algorithms are identical. Initially an approximated value for the quotient is determined then this approximated value is refined in an iterative manner, in each iteration the partial remainder is calculated exactly.

2.1.3.1 Accurate Quotient Approximation

Accurate Quotient Approximations is an iterative division algorithm based on lookup table that gives an approximation to the reciprocal of the divisor [39]. The algorithm iterates by using the reciprocal to find an approximated value for the quotient, then the resulted quotient is multiplied by the divisor, and finally the multiplication result is subtracted from the partial remainder to find the new partial remainder as follow,

$$Q_i = Q_{i-1} + R_{i-1}^h \times \frac{1}{Y_h} \times r^{-j} \tag{2.32}$$

$$R_i = R_{i-1} - R_{i-1}^h \times \frac{1}{Y_h} \times Y \tag{2.33}$$

Where X_h is defined as the most significant $m+1$ bits of the normalized dividend $X\,(n-\text{bit})$ and is extended by zeroes to form $(n-\text{bit})$ number, and Y_h is defined as the most significant m bits of normalized $Y(n-\text{bit})$ and is extended by ones to form $(n-\text{bit})$ number. Hence, X_h is always smaller than or equal to X, and Y_h is always larger than or equals to Y. Therefore the reciprocal approximation $\frac{1}{Y_h}$ is always smaller than or equal $\frac{1}{Y}$, then $\frac{X_h}{Y_h}$ is always smaller than or equal to $\frac{X}{Y}$.

The new quotient in each iteration is the result of adding the old quotient of the previous iteration and the approximated quotient in the current iteration; so the quotient is not guaranteed until the last iteration as it can be changed during the operation due to there is carry propagates through the quotient every iteration because of addition operation. For example let the intermediate result be $10000111\ldots11111$, this value can be changed in the next iteration because the approximated quotient will be added to it.

The algorithm is as follows,

1. Initially set $Q_0 = 0$, $R_0 = X$, and let variable $j = 0$ which represents number of bits retired every iteration.
2. Obtain an approximation of $\frac{1}{Y_h}$ from a look-up table using the most significant m-bit of Y. The width of the obtained approximation is m-bit and the approximation of $\frac{1}{Y_h}$ is normalized hence, the most significant bit is always one so m-1 bits will be stored and the most significant one will be concatenated.
3. The product $Y' = \frac{1}{Y_h} \times Y$ is invariant across all iterations, therefore, it needs to be performed once and all iterations will perform the multiplication $R_{i-1}^h \times Y'$ instead two successive multiplications $R_h \times Y$ then multiply the result by $\frac{1}{Y_h}$.
4. The product $R_{i-1}^h \times \frac{1}{Y_h}$ represents the new quotient digits.
5. Perform the product $R_{i-1}^h \times Y'$.
6. the next partial remainder is obtained as follow,

$$R_i = R_{i-1} - R_{i-1}^h \times Y' \tag{2.34}$$

7. The new quotient is then developed as follow,

$$Q_i = Q_{i-1} + R_{i-1}^h \times \frac{1}{Y_h} \times r^{-j} \tag{2.35}$$

8. Normalize the R_i by shit it to left in order to remove any leading zeros. The analysis of the algorithm will prove later that the algorithm guarantees removing $m - 2$ leading zeros every iteration.
9. The variable j is adjusted such that $j = j + m - 2$.
10. Steps from 4 to 9 are repeated until $j \geq n$, where n represents the quotient desired precision.

After the completion of all iterations, the top n bits of Q_n represent an approximation of the quotient accurate up to n bits. And the final remainder is determined by shifting R_j to the right by a value equal to j bits. In case of the most significant k bits of the quotient are used only the final remainder will be the result of adding $Q_{(j-k-1:\,0)} \times Y$ to $R_j \times 2^{-j}$.

2.1.3.2 Analysis of AQA

To determine how many bits are retired per iteration, it is required to determine the maximum value of new partial remainder R_i at iteration i; hence the number of leading zeroes will represent the number of bits retired in each iteration. All analysis will be done for general radix r then a substitution with binary radix.

By examining equation (2.31) which is used to calculate the new partial remainder there are two sources of error due to using the approximation $\frac{1}{Y_h}$. Let $\frac{1}{Y_h}$ be defined as,

$$\frac{1}{Y_h} = \frac{1}{Y} - E_a - E_b \tag{2.36}$$

Where E_a is an error due to using $\frac{1}{Y_h}$ instead of using $\frac{1}{Y}$ so this error is always non-negative as $\frac{1}{Y_h} \leq \frac{1}{Y}$, and E_b represents the error of storing $\frac{1}{Y_h}$ in finite number of bits, hence this error also is non-negative as $\frac{1}{Y_h}$ is truncated to be stored.

By substituting for $\frac{1}{Y_h}$ in equation (2.31),

$$R_i = R_{i-1} - R_{i-1}^h \times \left(\frac{1}{Y} - E_a - E_b\right) \times Y$$

$$R_i = (R_{i-1}^h + \Delta R_{i-1}^h) - R_{i-1}^h \times \left(\frac{1}{Y} - E_a - E_b\right) \times Y$$

$$\therefore R_i = \Delta R_{i-1}^h + R_{i-1}^h \times (E_a + E_b) \times Y \tag{2.37}$$

Where ΔR_{i-1}^{h} represents the difference between R_i and R_{i-1}^{h}, hence its value is always,

$$0 \leq \Delta R_{i-1}^{h} \leq \left(r^{-(m+1)} - r^{-n}\right) \tag{2.38}$$

, hence the maximum value will be

$$\Delta R_{i-1}^{h}\big|_{max} = \left(r^{-(m+1)} - r^{-n}\right) \tag{2.39}$$

The value of R_{i-1}^{h} is $0.1 \leq R_{i-1}^{h} \leq \left(r^{-1} - r^{-(m+1)}\right)$, so the maximum value of R_{i-1}^{h} is,

$$R_{i-1}^{h}\big|_{max} = \left(r^{-1} - r^{-(m+1)}\right) \cong 1 \tag{2.40}$$

As mentioned earlier E_a represents the error due to using $\frac{1}{Y_h}$ instead of using $\frac{1}{Y}$, hence E_a can be defined as,

$$E_a = \frac{1}{Y} - \frac{1}{Y_h} = \frac{Y_h - Y}{Y \times Y_h} \tag{2.41}$$

The normalized Y value is,

$$r^{-1} \leq Y \leq (1 - r^{-n}) \tag{2.42}$$

And the value of Y_h is,

$$(r^{-1} + r^{-m} - r^{-n}) \leq Y_h \leq (1 - r^{-n}) \tag{2.43}$$

Hence,

$$Y_{min} = r^{-1} \tag{2.44}$$

$$Y_h|_{min} = (r^{-1} + r^{-m} - r^{-n}) \tag{2.45}$$

$$(Y - Y_h)_{min} = 0.00 \ldots 00(r-1)_{m-1} \ldots (r-1)_{-n} = (r^{-m} - r^{-n}) \tag{2.46}$$

By substituting for Y and Y_h in equation (2.41),

$$\therefore E_a = \frac{(r^{-m} - r^{-n})}{r^{-1} \times (r^{-1} + r^{-m} - r^{-n})}$$

$$\therefore E_a < r^{-m+2} \tag{2.47}$$

As mentioned earlier E_b represents the truncation error of $\frac{1}{Y_h}$ so,

$$E_b = \frac{1}{Y_h}\Big|_{true} - \frac{1}{Y_h}\Big|_{stored} \tag{2.48}$$

Let $\frac{1}{Y_h}$ is stored in a table look-up with entry width $m-$ digit, and the stored digits are $d_0.d_{-1}d_{-2}d_{-3}d_{-4}\dots d_{-m+2}d_{-m+1}$, hence the least significant digit stored will be of weight r^{-m+1},

$$\therefore E_b < r^{-m+1} \tag{2.49}$$

By substituting for R_{i-1}^h , E_a , E_b , Y in equation (2.36), R_i will be,

$$R_i = \Delta R_{i-1}^h + R_{i-1}^h \times E_a \times Y + R_{i-1}^h \times E_b \times Y$$

$$R_i < r^{-m-1} - r^{-n} + r^{-m+2} \times r^{-1} + r^{-m+1} \times r^{-1}$$

$$\therefore R_i < r^{-m-1} + r^{-m+1} + r^{-m} \tag{2.50}$$

For $r = 2$

$$\therefore R_i < 2^{-m-1} + 2^{-m+1} + 2^{-m} = 1.75 \times 2^{-m+1}$$

$$\therefore R_i < 2^{-m+2} \tag{2.51}$$

Hence $m-2$ bits are guaranteed every iteration.

The previous analysis assumes that all operations are performed for full precision operands; however, if we consider that the operations used to calculate R_i are all of finite width then there may be other errors. For example if the multiplication of $\frac{1}{Y_h} \times Y$ to produce Y' is rounded or truncated, then the multiplication $R_{i-1}^h \times Y'$ is also rounded or truncated, then the subtraction $R_{i-1} - R_{i-1}^h \times Y'$ is again rounded or truncated. Then there will be a different total error depending on the locations of the rounding and the effect of this accumulation.

2.1.3.3 Short Reciprocal

Matula [40] presented the short reciprocal algorithm to achieve very high radix divider, short reciprocal algorithm is similar to AQA division algorithm to obtain a radix 2^{17} divider, instead of using different type multipliers, Matula used a single multiplier with length 18×69 and there is an additional adder-port that can perform a fused multiply/add operation. Therefore it can be used as a 19×69 multiplier.

The algorithm is performed as follow,

1. A low precision approximation for the reciprocal $\frac{1}{Y_h}$ is obtained from a lookup table, and then this seed is refined using two Newton-Raphson iterations which add more latency. The short reciprocal value used is slightly larger than the actual value of the reciprocal to insure that the quotient digit values are always equal to or greater by one unit in the last place than the exact values of the quotient digits. Hence any error will be equal to or greater than the unit in the last place of the quotient digit. This error can be eliminated by the succeeding quotient digit.

2. The algorithm uses equations (2.32) and (2.33) to obtain the new approximated quotient and the corresponding partial remainder. In contrast to AQA algorithm the multiplication $R_{i-1}^{h} \times \frac{1}{Y_h}$ is performed every iteration. The most significant 17 bits of the multiplication result represent the new quotient digit, so this value is truncated, and then this digit is shifted appropriately and accumulated to generate partial quotient and finally a full precision quotient. The multiplication $R_{i-1}^{h} \times \frac{1}{Y_h}$ is reused in equation (2.33) as it is multiplied by the divisor Y, and then the result is subtracted from the partial remainder R_{i-1} to produce new partial remainder R_i which is shifted to left to eliminate the leading zeroes where the count of these zeroes represents the radix of the division algorithm. The remainder may be negative if quotient digit is greater than the actual quotient digit.
3. If the final partial remainder is negative, the accumulated quotient is decremented by one unit in the last place, and the appropriately shifted divisor is added to the final partial remainder.

This scheme guarantees that 17 bits of quotient are retired in every iteration.

2.2 Decimal Division Algorithms

In the literature there are two main techniques followed to design decimal dividers, which are digit recurrence division technique and function iteration division technique. This section explains in brief the two dividers, the first one is based on SRT algorithm and the other is based on Newton-Raphson algorithm.

2.2.1 SRT-Based Decimal Divider

Tomas Lang and Alberto Nannarelli [41] presented a SRT decimal divider, where the iterative equation for division is derived from equation (2.3) as follow,

$$R_i = 10 \times R_{i-1} - q_i \times Y \tag{2.52}$$

To simplify the quotient digit selection function and generation of divisor multiples the authors split the quotient digit into two parts q_{Hi} and q_{Li} as shown in the following equation,

$$q_i = 5 \times q_{Hi} + q_{Li} \tag{2.53}$$

Where $q_{Hi} \in [-1, 0, 1]$, and $q_{Li} \in [-2, \ldots\ldots, 2]$, hence q_i ranges from -7 to 7 with a redundancy factor $\rho = \frac{5}{9}$.

By substituting q_i in equation (2.50) by equation (2.51), we will have,

$$R_i = (10 \times R_{i-1} - 5 \times q_{Hi} \times Y) - q_{Li} \times Y = V_i - q_{Li} \times Y \tag{2.54}$$

For quotient digit selection function, q_{Hi} is obtained by comparison of the truncated partial remainder with a truncated multiple of $5 \times Y$, then V_i is computed as follow,

$$V_i = 10 \times R_{i-1} - 5 \times q_{Hi} \times Y \tag{2.55}$$

A truncated value of V_i is compared with multiples of the divisor to obtain q_{Li} , then the new quotient digit is obtained according to equation (2.51) and the new partial remainder is calculated according to equation (2.52).

To avoid the generation of divisor multiples for every operation, the interval of the divisor is partitioned into sub-intervals $[Y_p, Y_{p+1})$ where $Y_{p+1} = Y_p + \Delta Y$, for sub-range p, and each sub-range contains m_{Hk} and m_{Lk} constants that are used to be compared with the truncated partial remainder and truncated value of V_i respectively. So that either the quotient digit q_{Hi} or q_{Li} equals k or j respectively according to the following equations,

$$m_{Hp}(k) \leq \widehat{10 \times R_{i-1}} < m_{Hp}(k+1) \tag{2.56}$$

$$m_{Lp}(j) \leq \widehat{V_{i-1}} < m_{Lp}(j+1) \tag{2.57}$$

2.2.2 Newton-Raphson Based Divider

Liang-Kai Wang, and Micheal Schulte (2004) [42], presented a new decimal division algorithm based on functional iteration method. The division algorithm is based on the calculation of the reciprocal of the divisor using the Newton–Raphson method, followed by a final multiplication by the dividend to obtain the quotient.

Here the dividend and the divisor are normalized operands such that, $0.1 \leq X < 1.0$ and $0.1 \leq Y < 1.0$ and $X < Y$, so the quotient result will be $0.1 < Q < 1$.

To determine the initial approximation of the divisor reciprocal, the divisor is divided into two parts $Y_M = 0.Y_{-1}Y_{-2}Y_{-3} \ldots Y_{-k}$, $Y_L = 0.Y_{-k-1}Y_{-k-2}Y_{-k-3} \ldots Y_{-n}$. Then the function $f(y) = \frac{1}{Y}$ is expanded around point $y = Y_M + 5 \times 10^{-k-1}$ using Taylor series expansion as follow,

$$f(y = A) \approx f(A) + f'(A) \times (y - A)$$

$$\therefore \frac{1}{Y} \approx \frac{1}{Y_M + 5 \times 10^{-k-1}} - \frac{Y - \left(Y_M + 5 \times 10^{-k-1}\right)}{(Y_M + 5 \times 10^{-k-1})^2}$$

$$\therefore \frac{1}{Y} \approx \frac{1}{(Y_M + 5 \times 10^{-k-1})^2} \times \left(Y_M - Y_L + 10^{-k}\right) \tag{2.58}$$

Let,

$$C' = \frac{1}{(Y_M + 5 \times 10^{-k-1})^2} \quad \text{and} \quad Y' = Y_M - Y_L + 10^{-k}$$

Then the most significant 2k digits of C' are obtained from a look-up table using the Y_M as the address of the lookup table, and Y' is calculated directly and then the most significant 2k

digits are kept. Then the multiplication $R_0 = C' \times Y'$ is performed and the most significant 2k-1 digits of the result are kept and the other digits are truncated. All these truncations will lead to introduce errors in which can be found in the following equation,

$$\therefore -0.55 \times 10^{-2k+3} < \varepsilon_{R_0} < 0 \tag{2.59}$$

After that, the initial approximation of the reciprocal R_0 is refined by performing m Newton-Raphson iterations as follow,

$$R_{i+1} = R_i \times (2 - Y \times R_i) \tag{2.60}$$

Which leads to enhance the approximated reciprocal error value after m iterations to be,

$$\therefore \left|\varepsilon_{R_m}\right| < 10^{-n-2} \tag{2.61}$$

Then an approximation for the quotient Q' is obtained from multiplying the dividend X by the approximated reciprocalR_m, then a truncated version Q' (most significant 2n-1 digits) of Q' is multiplied by the divisor Y , then the result of this multiplication is compared with the dividend X to obtain the sign of the remainder and to round the final quotient correctly.

2.3 Conclusion

This chapter presents different division algorithms that are used for binary division and how the researchers modified some of these algorithms such as digit recurrence and Newton-Raphson division algorithms to use them for decimal division.

The Radix-10 digit recurrence division algorithms are considered as digit by digit division algorithms as one quotient digit is retired every iteration. So for the quotient that required high precision it will take number of iterations equal to this precision. Also if it is required to increase the number of digits that retired every iteration the quotient digit selection function will be very complex. On the other hand, the convergence rate toward the quotient of decimal division that based on functional algorithms is quadratic. The main operations of functional algorithms are two sequential multiplications. The functional algorithms do not calculate the quotient directly, but refine an approximation of the reciprocal to the desired accuracy, and then a final decimal multiplication is required to produce the quotient.

The accurate quotient approximation can be considered as moderate algorithm between the digit recurrence algorithm and the functional algorithm. As large number of quotient digits are retired per iteration with simple operation. In the following chapters we are going to modify the accurate quotient approximation binary division algorithm to be used for decimal division. The details of implementing each block are given and a comparison between the synthesis results of the proposed divider with previous implemented dividers.

Chapter 3. Decimal Arithmetic Units

This chapter presents Binary Coded Decimal formats and some basic modules which are used in our proposed divider. The decimal coded formats will be $4221-\text{BCD}$ and $5211-\text{BCD}$. Also it presents three modules, which are Carry Extractor, BCD Adder, and Radix-10 Multiplier. These modules have been implemented in many previous designs [5] [9] [16-25]. Here we present specific ways to perform these three modules to enhance the performance of the divider.

3.1 BCD Formats

Equation 3.1 gives a formula for an integer decimal number D that consists of m-BCD digits and each digit $D_i \in \{0:9\}$, where i is a non-negative integer number represents the weight of the decimal digit. The decimal digits D_i can have different representations for the same value according to the weights of the bits $(r_3,\ r_2,\ r_1,\ r_0)$ within the decimal digit. The following equations show the relation among these elements.

$$D = \sum_{i=0}^{i=m-1} 10^i \times D_i \tag{3.1}$$

And

$$D_i = \sum_{j=0}^{j=3} r_j \times d_{ij} \tag{3.2}$$

Where r_j is the weight of bit j of the decimal digit, and d_{ij} is the value of the bit at weight j within digit D_i.

The most used binary coded decimal format is $8421-\text{BCD}$, which is a non-redundant format as each digit has only one representation. There are many other formats which can be used such as $4221-\text{BCD}$ and $5211-\text{BCD}$ [5]. These formats are redundant formats where two or more different representations exist for the same decimal digit. Table 3.1 shows the different representations for decimal digit D_i.

For $4221-\text{BCD}$ and $5211-\text{BCD}$ formats the sum of weight bits equals nine, and they are suitable for simple decimal carry save addition. There is another advantage that is all sixteen possibilities of 4-bit vector give a valid decimal number $D_i \in \{0:9\}$,

$$\sum_{j=0}^{j=3} r_j = 9 \tag{3.3}$$

The nines complement of any decimal number coded in $4221-\text{BCD}$ or $5211-\text{BCD}$ formats can be obtained by inverting all bits of the decimal number (one's complement), as,

$$9 - D_i = \sum_{j=0}^{j=3} r_j - \sum_{j=0}^{j=3} r_j \times d_{ij} = \sum_{j=0}^{j=3} r_j \times (1 - d_{ij}) = \sum_{j=0}^{j=3} r_j \times \overline{d_{ij}} \tag{3.4}$$

Hence we can obtain the negative value of any decimal number by inverting all bits of the number and then adding one as follows,

$$-D_i = 1 + 9 - \sum_{j=0}^{j=3} r_j \times d_{ij} = 1 + \sum_{j=0}^{j=3} r_j \times \overline{d_{ij}} \tag{3.5}$$

Digit	8421	4221	5211
0	0000	0000	0000
1	0001	0001	0001 or 0010
2	0010	0010 or 0100	0011 or 0100
	0011	0011 or 0101	0101 or 0110
4	010	1000 or 0110	0111
5	0101	1001 or 0111	1000
6	0110	1010 or 1100	1001 or 1010
7	0111	1011 or 1101	1011 or 1100
8	1000	1110	1110 or 1101
9	1001	1111	1111

Table 3.1: Different BCD Representations for a Decimal Number

We can obtain multiple 2D of any decimal number coded in $8421-\text{BCD}$ format as each $8421-\text{BCD}$ digit is first recoded to the (5211) decimal coding, and then left shift by one digit is performed to the recoded multiplicand, obtaining the $2D$ multiple in $4221-\text{BCD}$ [5].

$$2X_{4221} = L_{1-\text{bit shift}}(X_{5211}) \tag{3.6}$$

3.2 Proposed Carry Extractor

Carry extraction is a technique that is used to extract the carry out of two numbers without addition process also this technique can be used to enhance the addition process as we can avoid carry propagation by extracting the carry in to each digit of the sum using separate tree; the proposed process is performed in two steps. First we generate three flag vectors for the operands, the first vector is called propagate vector as each bit of this vector represent the logic XOR value of the corresponding operands bits. The second vector is called Generate vector as each bit of this vector represent the logic AND value of the corresponding operands bits The third vector is called OR vector as each bit of this vector represent the logic OR value of the corresponding operands bits. As shown in equations (3.7, 3.8, 3.9).

$$p_i = a_i \oplus b_i \tag{3.7}$$

$$g_i = a_i \,\&\, b_i \tag{3.8}$$

$$o_i = a_i + b_i \tag{3.9}$$

After that these three vectors are used to produce the digit level propagate and generate vectors for each corresponding digits of the of the operands as shown in equations (3.10, 3.11)

$$P_i = p_{i3}\overline{p_{i2}}\,\overline{p_{i1}}p_{i0} + \overline{o_{i3}}p_{i2}g_{i1}p_{i0} + \overline{o_{i3}}\,g_{i2}\overline{o_{i1}}p_{i0} \tag{3.10}$$

$$G_i = g_{i3} + p_{i3}(o_{i2} + o_{i1} + g_{i0}) + g_{i2}(o_{i1} + g_{i0}) + p_{i2}g_{i1}g_{i0} \tag{3.11}$$

Where,

$$\begin{cases} P_i = 1 & \text{when} \quad A_i + B_i = 9 \\ P_i = 0 & \text{o.w} \end{cases} \tag{3.12}$$

$$\begin{cases} G_i = 1 & \text{when} \quad A_i + B_i > 9 \\ G_i = 0 & \text{o.w} \end{cases} \tag{3.13}$$

The carry out of digit i is extracted according to the following equation,

$$C_{i+1} = \sum_{j=0}^{j=i} \left(g_j \prod_{k=j+1}^{k=i} p_k \right) \tag{3.14}$$

3.3 Proposed BCD Adder

Figure 3.1 shows the block diagram of the proposed 8421-BCD adder which is used to add two operands each of length N-digit and the two operands are coded in 8421 decimal format, this adder consists of Propagate-Generate-Or vectors extractor, One-Digit Hexadecimal Adder, Carry-in extractor, and Post Correction module.

At first we generate the three flag vectors propagate, generate, and or vectors for the operands, as shown in equations (3.7, 3.8, 3.9).

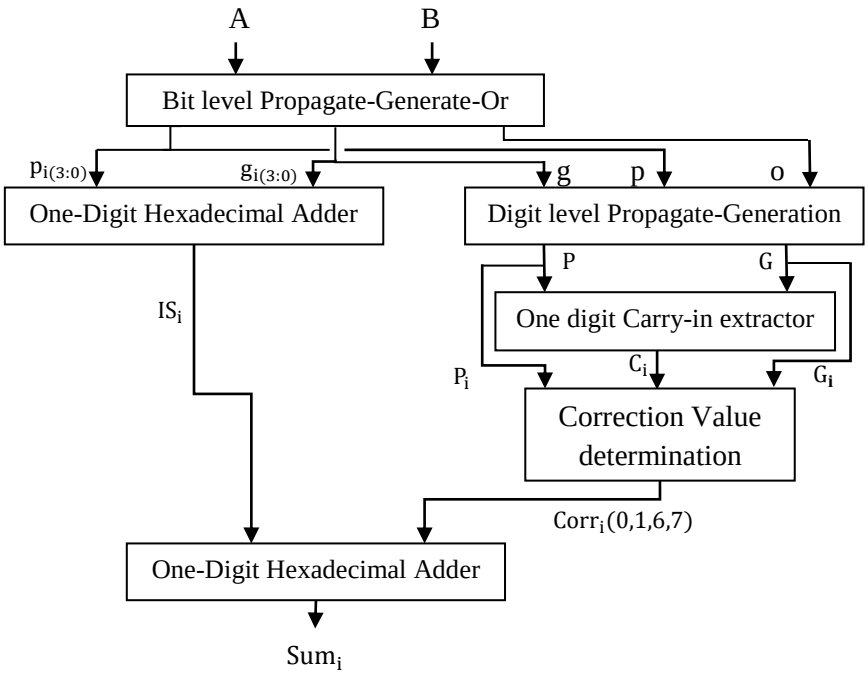

Figure 3.1: BCD adder

The two operands will be added to each other using carry-free hexadecimal adder as each digit of operand-A will be added to the corresponding digit of operand-B without any carry in using One-Digit Hexadecimal Adder module, so all digits will be added in parallel to produce the intermediate sum. Each four bit of the bit level propagate and generate vectors are used to produce the intermediate sum digit as shown in equations (3.15, 3.16, 3.17, 3.18).

Each digit of the two operands is added using hexadecimal adder, so it is required to correct the resulted digit by adding six to it, if its value exceeds nine, also it is required to add one to each digit if there is carry in to this digit. Hence, there are four values to be added to each digit of the intermediate sum as follow,

- Zero if the carry in equals zero and the digit value is lower than or equal to nine.
- One if the carry in equals one and the digit value is lower than or equal to nine.
- Six if the carry in equals zero and the digit value is greater than nine.
- Seven if the carry in equals one and the digit value is greater nine.

$$s_{i0} = p_{i0} \tag{3.15}$$

$$s_{i1} = p_{i1} \oplus g_{i0} \tag{3.16}$$

$$s_{i2} = p_{i2} \oplus (g_{i1} + g_{i0}p_{i1}) \tag{3.17}$$

$$s_{i3} = p_{i3} \oplus (g_{i2} + g_{i1}p_{i2} + g_{i0}\, p_{i1}p_{i2}) \tag{3.18}$$

So in parallel to the hexadecimal addition process we extract the digit level Propagate and Generate vectors for each digit of the intermediate sum as shown in equations (3.10, 3.11).

The carry in to each digit will be extracted according to equation (3.14),

Now after determining the value of each digit of the intermediate sum if it is equals nine or greater and the carry in to each digit, we can produce the value which will be added to each digit to produce the correct final sum as follow,

$$\begin{cases} (\text{Correction value})_i = 0 & \text{when} \quad C_i = 0 \text{ and } G_i = 0 \\ (\text{Correction value})_i = 1 & \text{when} \quad C_i = 1 \text{ and } (G_i = 0 \text{ or } P_i = 0) \\ (\text{Correction value})_i = 6 & \text{when} \quad C_i = 0 \text{ and } G_i = 1 \\ (\text{Correction value})_i = 7 & \text{when} \quad C_i = 1 \text{ and } (G_i = 1 \text{ or } P_i = 1) \end{cases} \tag{3.19}$$

Again we will use the One-Digit Hexadecimal Adder module to add each digit of the intermediate sum to its correction value to produce the final correct sum.

3.4 Proposed Radix-10 Multiplier

Figure3.2 shows the block diagram of the proposed Radix 10 multiplier. The multiplier consists of: generation of multiplicand multiples, generation of partial products and reduction of partial products tree into two final vectors (M_1 and M_2).

Each digit of the multiplier $Y\,(n-\text{digit})$ will control a multiplexer to select the multiplicand $X\,(m-\text{digit})$ multiple out of $\{0, X, 2X, \ldots, 9X\}$ to determine the corresponding

partial product vector for each multiplier digit. Hence there will be (n) partial product vectors of length (m + 1 digit). Before the reduction of the partial product tree each partial product vector will be aligned according to the decimal weight of the multiplier selector digit. Then using carry save adder reduction tree the partial product tree will be reduced into two vectors each of length (2m+1).

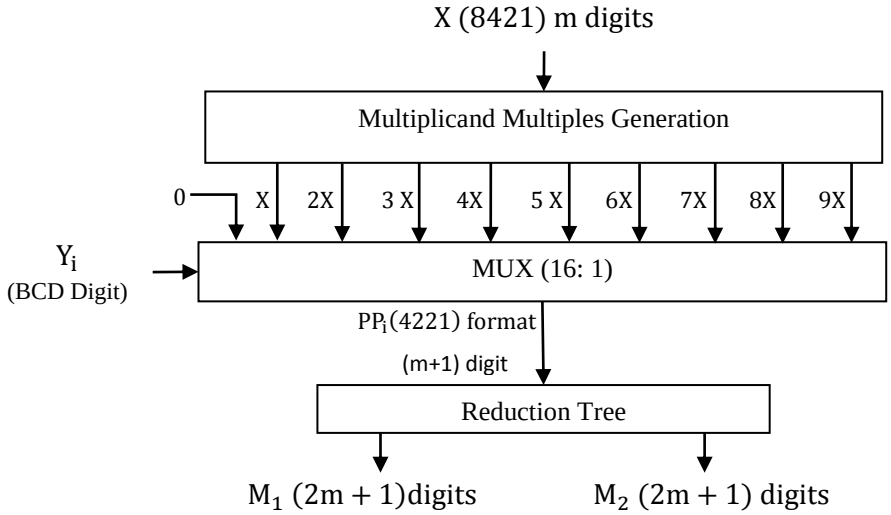

Figure 3.2: Radix 10 multiplier

3.4.1 Multiplicand Multiples Generation

It is required to generate all multiplicand multiples $\{0, X, 2X, \dots, 9X\}$, which is coded in $8421-$BCD format, and all generated multiples will be coded in $4221-$BCD format.

For the multiplicand if it is multiplied by any digit $\in \{2:9\}$ there will be two vectors to represent the multiple, the first vector is the individuals vector and the other is the tens vector. As when multiplying every digit of the multiplicand by any digit $\in \{2:9\}$ this will result in a number which consists of two digits according to table (3.2), and then we will form two vectors one of individuals and the other of tens according the weight of the multiplicand digit.

First we will generate multiples {2X, 3X, 5X, 6X, 7X, 9X}, then the multiple {4X} will be generated by multiplying multiple {2X} by two and the same procedure will be used to generate the multiple {8X} from multiple {4X}.

Till now for each multiple {2X, 3X, 5X, 6X, 7X, 9X}, we have two vectors which will be added to each other to produce the final multiple. But the summation of the maximum individual digit and the maximum tens digit of multiples {2X, 5X} doesn't exceed nine, hence when adding their vectors to each there will be no carry propagation and no correction will be needed to the final result.

For multiple {2X} as shown in table (3.2) the ten digit is always zero or one and the least significant bit of the individual digit is always zero because the result of multiplying by two is always even, hence the multiple {2X} can be obtained according to equations (3.20, 3.21, 3.22, 3.23), and the result multiple is in 4221-BCD format.

$$\{2X\}_{i0} = x_{(i-1)3} + x_{(i-1)2}x_{(i-1)1} \quad (3.20)$$

$$\{2X\}_{i1} = x_{i3} + x_{i2}\overline{x_{i0}} + \overline{x_{i2}}x_{i0} \quad (3.21)$$

$$\{2X\}_{i2} = x_{i3}x_{i0} + x_{i2}\overline{x_{i1}}\,\overline{x_{i0}} \quad (3.22)$$

$$\{2X\}_{i3} = x_{i3} + \overline{x_{i2}}x_{i1} \quad (3.23)$$

, where $0 \leq i \leq N + 1$

Multiplicand Digit \ Multiplier	2	3	5	6	7	9
0	0 0	0 0	0 0	0 0	0 0	0 0
1	0 1	0 3	0 5	0 6	0 7	0 9
2	0 4	0 6	1 0	1 2	1 4	1 8
3	0 6	0 9	1 5	1 8	2 1	2 7
4	0 8	1 2	2 0	2 4	2 8	36
5	1 0	1 5	2 5	3 0	3 5	4 5
6	1 2	1 8	3 0	3 6	4 2	5 4
7	1 4	2 1	3 5	4 2	4 9	6 3
8	1 6	2 4	4 0	4 8	5 6	7 2
9	1 8	2 7	4 5	5 4	6 3	8 1

Table 3.2: Multiples of single decimal digit

To obtain multiple {4X}, the multiple {2X} is recoded from 4221-BCD format to 5211-BCD format which is also redundant format, and then $L_{1-\text{bit shift}}$ is performed to the recoded multiplicand, obtaining the {4X} multiple in $4221-\text{BCD}$ as shown in equation (3.6). The multiple {8X} can be obtained from multiple {4X} by the same way.

Multiple {5X}, as shown in table (3.2) the individual digit is always zero or five if the corresponding multiplicand digit is even or odd. So we can merge the individual of digit X_i with the ten of digit X_{i-1} to obtain the multiple {5X} which is recoded in 4221-BCD format directly according to equations (3.24, 3.25, 3.26, and 3.27).

$$\{5X\}_{i0} = x_{i0} \oplus x_{(i-1)0} \quad (3.24)$$

$$\{5X\}_{i1} = x_{(i-1)1} + x_{i0}\,x_{(i-1)0} \quad (3.25)$$

$$\{5X\}_{i2} = x_{(i-1)2} \quad (3.26)$$

$$\{5X\}_{i3} = x_{i0} \quad (3.27)$$

For multiples {2X, 5X} the summation of the maximum individual digit and the maximum tens digit doesn't exceed nine, hence when adding their vectors to each there will be no carry propagation and no correction to the final result will be needed, so we will use two rows of One-Digit Hexadecimal Adder, one is used to add multiple {2X} vectors and the other to add multiple {5X} vectors.

The individual and ten digits of multiples {3X, 6X, 7X, 9X} are obtained as shown in table 3.3.

Then each digit of the individuals and the tens are aligned according to multiplicand digit X_i to form the individuals vector and the tens vector for each multiple, after that each corresponding two vectors are added using 8421-BCD adder to produce multiples {3X, 6X, 7X, 9X}.

Position / Multiple	Ten	Individual
3X	$ten_{i0} = x_{i2}\overline{x_{i1}} + x_{i2}\overline{x_{i0}}$ $ten_{i1} = x_{i3} + x_{i2}x_{i1}\,x_{i0}$ $ten_{i2} = 0$ $ten_{i3} = 0$	$ind_{i0} = x_{i0}$ $ind_{i1} = x_{i2}\overline{x_{i1}}\,\overline{x_{i0}} + \overline{x_{i2}}\,x_{i1}\overline{x_{i0}} + \overline{x_{i2}}\,\overline{x_{i1}}x_{i0}$ $ind_{i2} = x_{i3} + \overline{x_{i2}}\,x_{i1}\overline{x_{i1}} + x_{i2}\overline{x_{i1}}\,x_{i0}$ $ind_{i3} = \overline{x_{i2}}\,x_{i1}\,x_{i0} + x_{i2}\,x_{i1}\overline{x_{i0}}$
6X	$ten_{i0} = x_{i3}\,x_{i0} + \overline{x_{i2}}\,x_{i1} + x_{i2}\,\overline{x_{i1}}\,x_{i0} + x_{i1}\overline{x_{i0}}$ $ten_{i1} = x_{i2}\,\overline{x_{i1}} + x_{i2}\overline{x_{i0}}$ $ten_{i2} = x_{i3} + x_{i2}x_{i1}x_{i0}$ $ten_{i3} = 0$	$ind_{i0} = 0$ $ind_{i1} = \overline{x_{i3}}\,\overline{x_{i2}}\,\overline{x_{i1}}\,x_{i0} + x_{i2}x_{i1} + x_{i1}\overline{x_{i0}}$ $ind_{i2} = x_{i2}\overline{x_{i0}} + \overline{x_{i2}}\,\overline{x_{i1}}\,x_{i0}$ $ind_{i3} = x_{i3}\overline{x_{i0}} + \overline{x_{i2}}\,x_{i1}x_{i0}$
7X	$ten_{i0} = x_{i3}\,\overline{x_{i0}} + \overline{x_{i2}}\,x_{i1}\overline{x_{i0}} + x_{i2}\,\overline{x_{i1}}\,x_{i0}$ $ten_{i1} = x_{i2}x_{i0} + x_{i2}\,\overline{x_{i1}} + \overline{x_{i2}}\,x_{i1}\,x_{i0}$ $ten_{i2} = x_{i3} + x_{i2}x_{i1}$ $ten_{i3} = 0$	$ind_{i0} = x_{i0}$ $ind_{i1} = x_{i3} + \overline{x_{i2}}\,\overline{x_{i1}}\,x_{i0} + x_{i2}x_{i1}\overline{x_{i0}}$ $ind_{i2} = x_{i3}\overline{x_{i0}} + \overline{x_{i3}}\,\overline{x_{i1}}\,x_{i0} + \overline{x_{i2}}x_{i1}\overline{x_{i0}}$ $ind_{i3} = x_{i2}\,\overline{x_{i1}}\,\overline{x_{i0}} + x_{i2}\,x_{i1}x_{i0}$
9X	$ten_{i0} = x_{i3}\,\overline{x_{i0}} + x_{i2}\overline{x_{i0}} + x_{i1}\,\overline{x_{i0}}$ $ten_{i1} = x_{i3}\,\overline{x_{i0}} + x_{i2}\,\overline{x_{i1}}\,\overline{x_{i0}} + x_{i1}\,x_{i0}$ $ten_{i2} = x_{i3}\,\overline{x_{i0}} + x_{i2}x_{i1} + x_{i2}x_{i0}$ $ten_{i3} = x_{i3}x_{i0}$	$ind_{i0} = x_{i0}$ $ind_{i1} = x_{i3}\overline{x_{i0}} + x_{i2}\overline{x_{i1}}\,\overline{x_{i0}} + x_{i1}x_{i0}$ $ind_{i2} = x_{i2}\overline{x_{i1}} + x_{i2}\overline{x_{i0}} + \overline{x_{i2}}x_{i1}x_{i0}$ $ind_{i3} = \overline{x_{i3}}\,\overline{x_{i2}}\,\overline{x_{i1}}\,x_{i0} + \overline{x_{i2}}\,x_{i1}\overline{x_{i0}}$

Table 3.3: The individual and ten digits of multiples {3X, 6X, 7X, 9X}

3.4.2 Partial Product Array

As described before, the radix-10 architecture produces n partial products vectors coded in (4221) format. Before being reduced, the n partial products (PP^i) are aligned according to their decimal weights by $4i-$ bit wired left shifts ($PP^i \times 10^i$). Figure 3.3 shows the resultant partial product array for 8×8 multiplier , so the number of partial product vectors is 8 vectors and it is required to reduce these vectors into two vectors only using 3:2 CSA's.

$$
\begin{array}{cccccccccccccccc}
 & & & & & & pp_8^0 & pp_7^0 & pp_6^0 & pp_5^0 & pp_4^0 & pp_3^0 & pp_2^0 & pp_1^0 & pp_0^0 \\
 & & & & & pp_8^1 & pp_7^1 & pp_6^1 & pp_5^1 & pp_4^1 & pp_3^1 & pp_2^1 & pp_1^1 & pp_0^1 & \\
 & & & & pp_8^2 & pp_7^2 & pp_6^2 & pp_5^2 & pp_4^2 & pp_3^2 & pp_2^2 & pp_1^2 & pp_0^2 & & \\
 & & & pp_8^3 & pp_7^3 & pp_6^3 & pp_5^3 & pp_4^3 & pp_3^3 & pp_2^3 & pp_1^3 & pp_0^3 & & & \\
 & & pp_8^4 & pp_7^4 & pp_6^4 & pp_5^4 & pp_4^4 & pp_3^4 & pp_2^4 & pp_1^4 & pp_0^4 & & & & \\
 & pp_8^5 & pp_7^5 & pp_6^5 & pp_5^5 & pp_4^5 & pp_3^5 & pp_2^5 & pp_1^5 & pp_0^5 & & & & & \\
pp_8^6 & pp_7^6 & pp_6^6 & pp_5^6 & pp_4^6 & pp_3^6 & pp_2^6 & pp_1^6 & pp_0^6 & & & & & & \\
pp_8^7 & pp_7^7 & pp_6^7 & pp_5^7 & pp_4^7 & pp_3^7 & pp_2^7 & pp_1^7 & pp_0^7 & & & & & &
\end{array}
$$

Figure 3.3: partial product array for 8×8 multiplier

3.4.3 Partial Product Reduction

As mentioned earlier all partial product array elements are recoded in 4221 redundant format, which simplify the process of PP reduction. Assume there are three decimal digits $D1_i, D2_i, D3_i$ coded in 4221 format, we can use binary carry save adder to add these three digits to each other which produces only two digits the sum (S) and the carry (H) both are recoded in 4221 format, according equation (3.28).

$$D1_i + D2_i + D3_i = \sum_{j=0}^{3} (d1_i + d2_i + d3_i) r_j$$

$$= \sum_{j=0}^{3} z_{i,j} \times r_j + 2 \times \sum_{j=0}^{3} h_{i,j} \times r_j$$

$$\therefore D1_i + D2_i + D3_i = Z_i + 2 \times H_i \qquad (3.28)$$

However it is required to multiply the carry digit (H) by two, so that the carry digit (H) is first converted to 5211 format and shifted to the left by 1-bit, then the result will be 2H coded in 4221 format.

In order to reduce the partial product array in Figure 3.3, we can apply equation (3.28) for decimal vectors instead of single decimal digits, so we can reduce every three vectors of the partial products array into two vectors using CSA according as shown in figure 3.4. Hence, for N vectors passed through one step of 3:2 CSA the number of resulted vectors will be $\left(\left\lfloor \frac{N}{3} \right\rfloor * 2\right) + 1 \times \text{remainder}\left(\left\lfloor \frac{N}{3} \right\rfloor\right)$ vectors, and then we apply the same operation on the resulted vectors until we reach to two vectors.

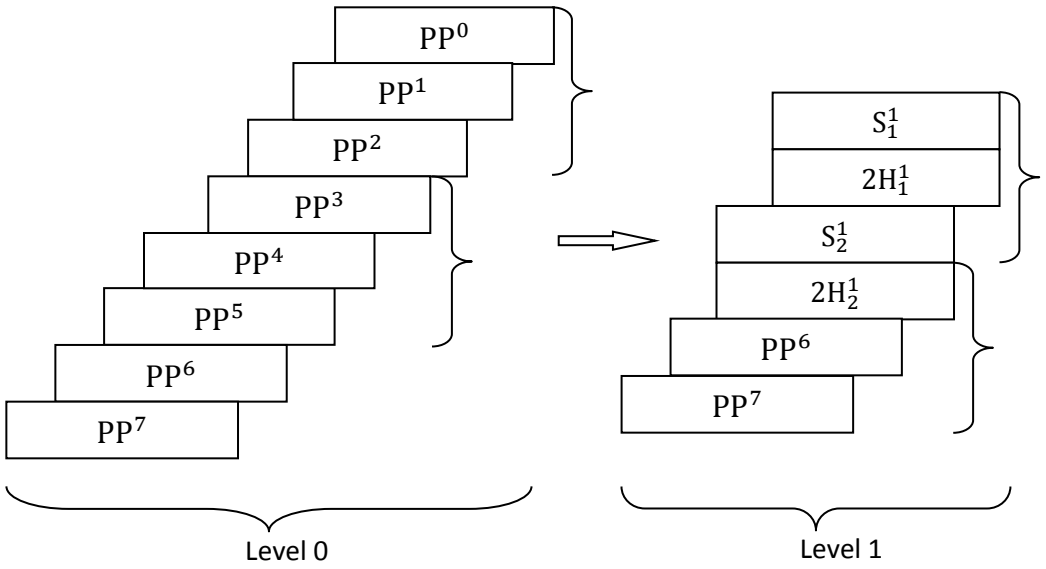

Figure 3.4: One step of partial product array reduction

To reduce three vectors firstly we use CSA stage to produce sum and carry vectors each of them is recoded in 4221 format then to multiply the carry vector by two each digit is first recoded to the (5211) decimal coding , then $L_{1-\text{bit shift}}$ is performed to the recoded multiplicand, which produce 2H multiple in 4221-BCD.

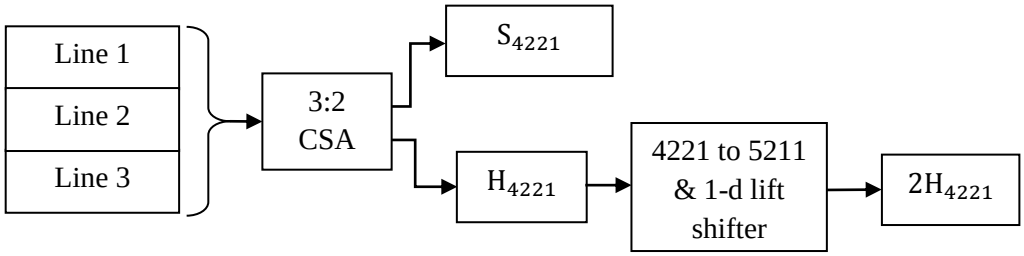

Figure 3.5: Reduction of three 4221-BCD vectors using CSA

At the end of partial product reduction stage there are two vectors obtained, the sum vector (M_1) and the carry vector (M_2). Both vectors are coded in the 4221 format.

In the next chapter the proposed Decimal Floating Point Divider is presented and all details of its design show that how we use the previously described modules.

Chapter 4. Design of the proposed DFP Divider

As presented in chapter 1, IEEE 754-2008 defines three formats for decimal floating point numbers which are decimal-32, decimal-64, and decimal-128. The proposed very high radix divider is implemented based on decimal-64 format. As the most used decimal dividers are based on decimal-64 format and, a divider based on decimal-128 format can be easily obtained by some modification of decimal-64 divider.

Figure 4.1 shows the proposed decimal floating point divider. As described in chapter 1, the decimal floating point number consists of three parts which are the sign, coefficient, and the exponent. Let the dividend be FX and the divisor be FY, and they are formed as follow,

$$FX = (-1)^{Sx} \times X \times 10^{Ex+bias} \tag{4.1}$$

$$FY = (-1)^{Sy} \times Y \times 10^{Ey+bias} \tag{4.2}$$

The coefficients X and Y represent the fractional numbers, these coefficient are not normalized as the fraction point is a virtual point at the right of the least significant digit of the coefficient, which is recommended by IEEE 754-2008 standard to keep trailing zeros, which help in many fields as described in 1.2.1 and 1.2.2.

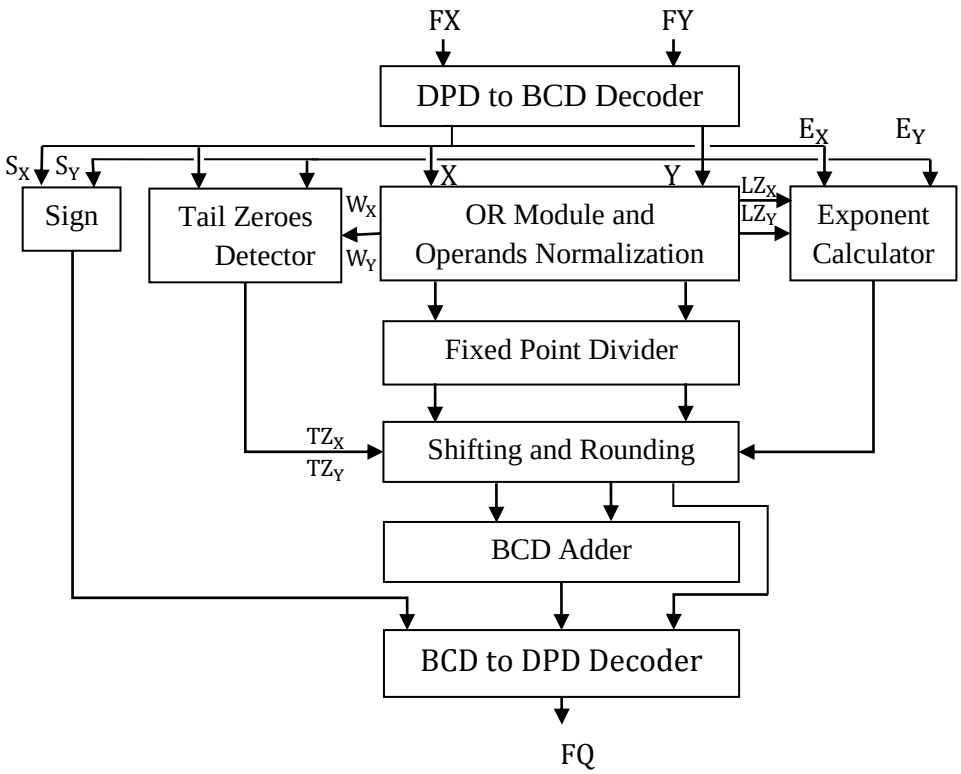

Figure 4.1: Architecture of IEEE 754-2008 divider

Because the coefficients X and Y are not normalized, they may contain leading zeros; these leading zeros should be removed via a shifter which will affect the calculation of the

exponent. Assuming LZ_X and LZ_Y are the leading zeros value of X and Y respectively, the exponent is calculated after shifting as follow,

$$E_Q = E_X - E_Y - LZ_X + LZ_Y + \text{Bias} \tag{4.3}$$

The normalized X and Y are the input dividend and divisor operands to the fixed point divider which produces three digits in each iteration. And the final quotient is normalized to left so it may be required to apply shift to right to the quotient, which will affect the final exponent value also. Shifting and rounding operations proceed on the final quotient to generate the final result compliant with IEEE 754-2008 standard.

4.1 Proposed Division Algorithm

This thesis presents an arithmetic algorithm and hardware design for decimal floating-point divider that is based on “Very High Radix” division algorithm [32]. Here we adopt "Accurate Quotient Approximations” binary division algorithm [39] and modify it to be used for decimal division.

Our contribution is to design a new decimal divider based on "Accurate Quotient Approximations” division algorithm and make all other operation for this divider to be compliant with IEEE 754-2008 as described later.

4.1.1 Accurate Quotient Approximations

Here we will modify binary Accurate Quotient Approximations division algorithm instead of using binary radix we use decimal radix, so the division iterative equations will be,

$$Q_i = Q_{i-1} + R_{i-1}^h \times \frac{1}{Y_h} \times 10^{-j} \tag{4.4}$$

$$R_i = R_{i-1} - R_{i-1}^h \times \frac{1}{Y_h} \times Y \tag{4.5}$$

Where R_i is the partial remainder after iteration i and initially equals the dividend X, R_{i-1}^h is defined as the $m+1$ most significant digits of R_i and is extended by zeros to form $(n-\text{digit})$ number, and Y_h is defined as the m most significant digits of Y and is extended by nines to form $(n-\text{digit})$ number. This leads to R_{i-1}^h being always smaller than or equals to R_i and Y_h is always greater than or equals to Y. Therefore the reciprocal approximation $\frac{1}{Y_h}$ is always smaller than or equals $\frac{1}{Y}$, hence $\frac{X_h}{Y_h}$ is always smaller than or equals $\frac{X}{Y}$.

Then we follow the same steps for AQA binary division algorithm as described earlier in chapter two (2.2.3.1).

After the completion of all iterations, the top n digits of Q form the true quotient. Similarly, the final remainder is formed by right-shifting R by $m-2$ digits. This remainder, though, assumes the use of the entire value of Q as the quotient. If only the top n-bits of Q are used as the quotient, then the final remainder is calculated by adding $Q_l \times Y$ to R, where Q_l comprises the low order bits of Q after the top n-bits.

To determine number of digits retired per iteration we will follow the same analysis for AQA binary division algorithm, by substituting in equation (2.50) for radix $r = 10$,

$$\therefore R_i < 10^{-m-1} + 10^{-m+1} + 10^{-m} \cong 1.21 \times 10^{-m+1} \tag{4.6}$$

$$\therefore R_i < 10^{-m+2} \tag{4.7}$$

Hence $m - 2$ digits are guaranteed every iteration.

4.2 Architecture

This part shows the overall architecture of the proposed AQA divider.

4.2.1 Operands Normalization

This module is used to normalize the dividend and the divisor by shifting each operand to left with a value equal to the number of the leading zeroes in each operand. In order to perform the shift operation we extract two flag words W_x and W_y of the dividend and the divisor as each bit represent the value of the corresponding digit if it is zero or not as shown in figure 4.2. Hence we will have W_x and W_y each representing the status of digits X and Y respectively which are used as control signals for two multiplexers to produce the dividend and the divisor in normalized form and the value of the shift for both operands X_{LZV} and Y_{LZV}. If one of the operands is zero the corresponding zero flag X_{ZF} and Y_{ZF} will be signaled.

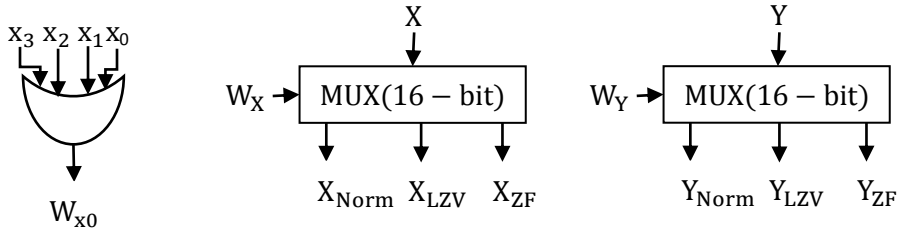

Figure 4.2: Extraction of zero flag vector for both operands

4.2.2 Intermediate Exponent Calculator

Here we calculate the exponent which may be adjusted later as mentioned earlier as any shift to the final quotient affect the final exponent value. This module will generate two flags the overflow flag will be generated if the intermediate exponent is larger than maximum exponent, and the underflow flag which will be generated if the intermediate exponent is lower than minimum exponent. Also it generates the under-flow value which will be used to shift the quotient to right to achieve right result.

There are two types of the exponent as follow, when both operands the dividend and the divisor are normal non-zero numbers, here we add the exponent of the dividend to the leading zeroes value of the divisor and the exponent of the divisor to the leading zeroes value of the dividend. Then we subtract the second result from the first result which will generate the exponent without the bias as shown in equation 4.3. Then we add the bias to the resultant exponent to generate the intermediate exponent as shown in figure 4.3.

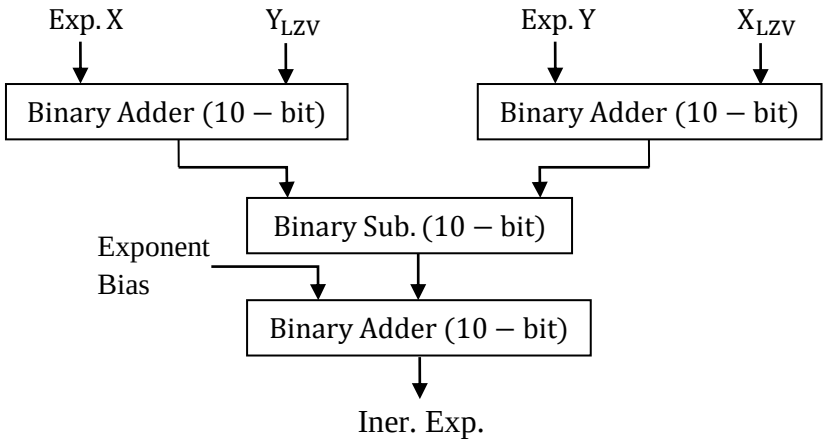

Figure 4.3: Intermediate normal exponent calculator

If the dividend is zero and the divisor is normal non-zero number the divisor exponent is subtracted from the dividend exponent directly and then the bias is added to the result to produce the intermediate exponent as shown in figure 4.4.

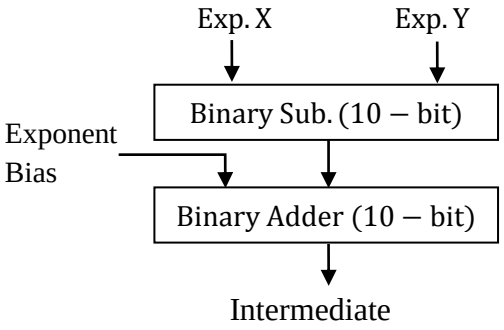

Figure 4.4: Intermediate exponent calculator for zero dividend

4.2.3 Decimal Fixed Point Divider

Figure 4.4 shows the block diagram of decimal fixed point divider which is the core of our decimal floating divider as normalized X is divided by normalized Y using AQA Division Algorithm. It is an iterative algorithm which can be represented by the following pseudo code,

$$\text{For } i = 0 : k$$

$$Q_i = Q_{i-1} + R_{i-1}^{h} \times \frac{1}{Y_h} \times 10^{-3i}$$

$$R_i = R_{i-1} - R_{i-1}^{h} \times Y'$$

$$\text{End}$$

Where (k) represents the number of desired iterations, $Y' = Y \times \frac{1}{Y_h}$, $R_0 = X$,

$R_h = R_{15}R_{14} \dots R_{m+1}\,00 \dots 00$, and $Y_h = Y_{15}Y_{14} \dots Y_m 99 \dots 99$.

The number of iterations is determined by the number of digits which are retired every iteration and the desired precision. The number of digits which are retired every iteration depends on the number of most significant digits of X and Y which are used to form X_h and Y_h respectively. Here we use the most significant five digits of X to form X_h and the most significant four digits of Y to form Y_h hence; the number of digits which retired every iteration will be three digits and to prove that we will use the following equation,

$$R_i = \Delta R_{i-1}^h + R_{i-1}^h \times E_a \times Y + R_{i-1}^h \times E_b \times Y \tag{4.8}$$

And then substituting for the maximum value of each element at $r = 10, Y = 10^{-1}$, $n = 16$, and $m = 4$ is,

$$\Delta R_{i-1}^h|_{max} = 10^{-5} - 10^{-16}$$

$$R_{i-1}^h|_{max} = 1 - 10^{-5}$$

$$E_a|_{max} = \frac{10^{-4} - 10^{-16}}{1 + 10^{-3} - 10^{-15}}$$

$$E_b|_{max} = 10^{-3}$$

$$\therefore R_i = 0.12 \times 10^{-3} \tag{4.9}$$

$$\therefore R_i < 10^{-3} \tag{4.10}$$

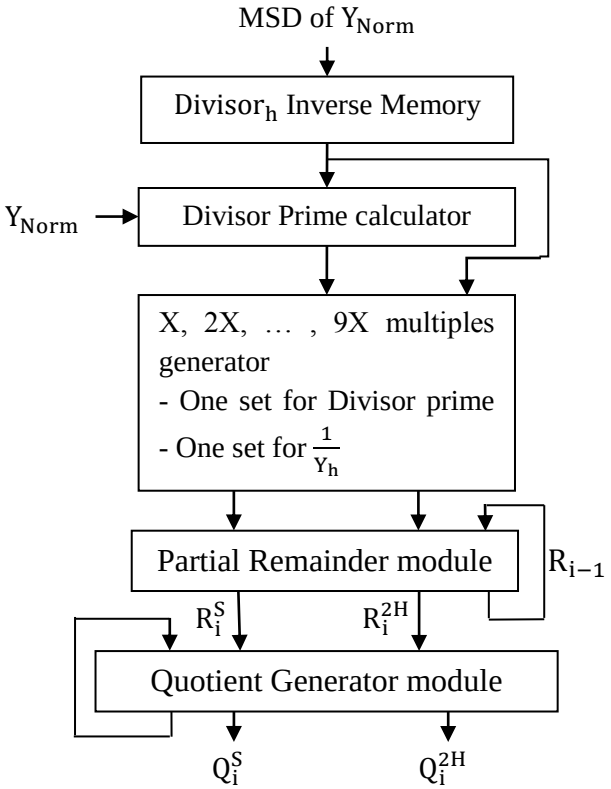

Figure 4.5: Fixed point divider

To carry out this operation three operands are used which are, X_{Norm}, Y_{Norm}, and $\frac{1}{Y_h}$.

An approximated value of $\frac{1}{Y_h}$ is stored in decimal form, in a binary addressed memory, accessed by conversion of the most significant four digits of $Y_{Norm}$to its binary value.

4.2.3.1 Divisor High Inverse Memory

An approximated value for $\frac{1}{Y_h}$ is obtained by accessing a memory using most significant four digits of normalized divisor ($Y_{15}Y_{14}Y_{13}Y_{12}$) as the address. This approximated value is the most significant four digits of $\frac{1}{Y_h}$. Instead of using this address in BCD form we convert this address form to Binary form to save memory area. So then we access the memory using the binary address.

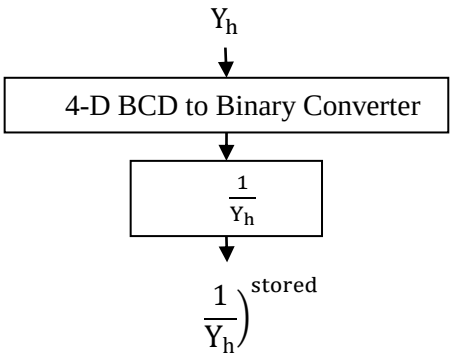

Figure 4.6: Divisor high inverse memory block

4.2.3.2 4-D BCD to Binary Converter

Here we convert each digit of the four digits decimal number to its corresponding binary value using stored values according to the weight of the decimal digit as follow,

Example 4.1 , Let $(B_{15}B_{14}B_{13}B_{12})_{bcd} = 1534$ Then,

B_{bin}^{15}	=	0001111101000	as	B_{bcd}^{15}	=	1000	,thousands	13 bit
B_{bin}^{14}	=	0100101100	as	B_{bcd}^{14}	=	500	,hundreds	10 bit
B_{bin}^{13}	=	0011110	as	B_{bcd}^{13}	=	30	,tens	7 bit
B_{bin}^{12}	=	0100	as	B_{bcd}^{12}	=	4	,individuals	4 bit

Then we add these four binary values to each other by dividing them into two parts most significant part and least significant part each is seven bits.

By observation we can note that the least significant three bits of the thousands binary number are always zero so we can occupy these three bits by the least significant three bits of the individual binary number. Also the least significant two bits of the hundreds binary number are zeros so we can represent the individual binary number when it equals eight or

nine by setting the least significant three digits of the thousands binary number to one and set the least significant two bit of the hundreds binary number to have value one or two if the individual binary number equals eight or nine respectively.

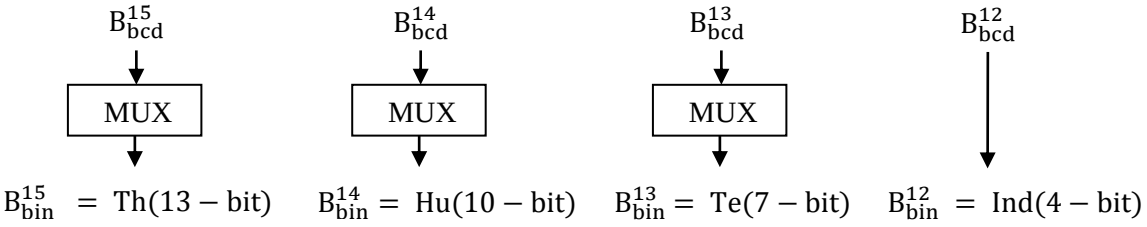

Figure 4.7: Divisor high digits conversion from BCD to Binary

Hence we will have three binary numbers which will be reduced to two numbers using CSA; and then these two binary numbers will be added to each other using CPA to produce the memory address as shown in figure 4.8.

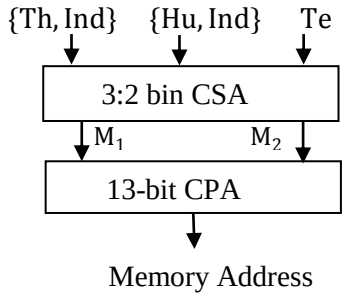

Figure 4.8: Binary memory address generation

4.2.3.3 $\frac{1}{Y_h}$ Memory

It is a memory which stores the most significant four digits of $\frac{1}{Y_h}$ in BCD form. This memory is accessed using the binary address generated in the previous stage.

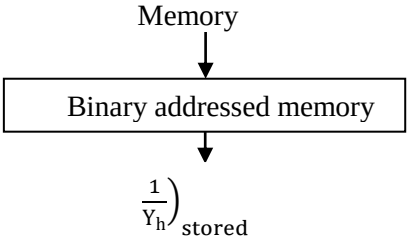

Figure 4.9: Memory access using binary address

Example 4.2, $Y_h = 1.534999999999999 \longrightarrow \frac{1}{Y_h} = 6.514657980456030302$

$$\therefore \frac{1}{Y_h})_{stored} = 6.514$$

4.2.3.4 Divisor Prime Calculator

Here we calculate divisor prime value once and then this value is used along the operation as this value is invariant. Divisor prime value is the result of multiplying Y_{Norm} by $\frac{1}{Y_h}\Big)_{stored}$, hence we reuse 38×16 Radix-10 Multiplier to perform this multiplication operation; all details of this module will be described later in 4.3.3.6.

The output of the multiplier will be two vectors which are the sum and carry representing the divisor prime. Then these two vectors are added to each other using BCD adder to produce the divisor prime.

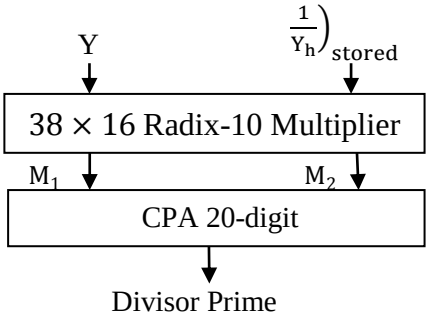

Figure 4.10: Divisor prime generation

4.2.3.5 Divisor Prime and Divisor High Inverse Multiples

According to our algorithm the value of Y' and $\frac{1}{Y_h}$ are invariant along the division operation hence instead of generation their multiples every iteration for the multiplication process we generate them once before entering the loop then we use these values along all iterations as shown in figure (4.4). To produce the multiples of the two numbers Y' and $\frac{1}{Y_h}$ we follow the technique that described in 3.4.1.

4.2.3.6 Partial Remainder Module

Figure 4.11 shows the block diagram of the partial remainder module, which is designed to execute equation 4.1. In order to generate the new partial remainder, the most significant six digits of partial remainder which represent the partial remainder high R_h, multiplied by divisor prime, then the result is subtracted from current partial remainder to generate the new partial remainder as shown in equation 4.1, at first iteration the partial remainder is the dividend.

The partial remainder high R_i^h will be used as a multiplier for the following multiplication operation, $R_i^h \times Y^{'}$.

For this multiplication operation the multiplicands multiples $(Y', 2Y', \dots, 9Y')$ have been generated before starting the fixed division operation as described in 4.3.3.5, in order to reduce the time of the multiplication process. Then each digit of the multiplier R_h^i will choose the corresponding multiples of the multiplicands Y' which produce six partial product vectors that

are reduced using CSA reduction tree to two vectors which are IR_h^S and IR_h^{2H} then these two vectors are added to each other using BCD adder to produce the new partial remainder.

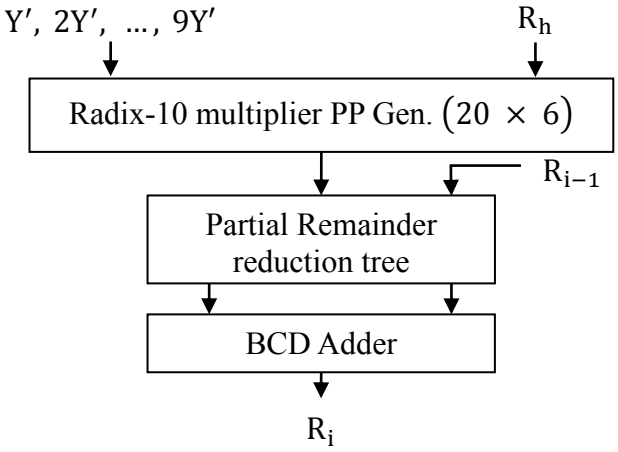

Figure 4.11: Partial remainder module

4.2.3.7 Quotient Generator Module

As shown in figure 4.12 the same steps that used to generate the new partial remainder are used to produce the new quotient digits as the partial remainder high is multiplied by the divisor high inverse $1/Y_h$, where the multiplier is the partial remainder high R_i^h the multiplicands multiples ($\frac{1}{Y_h}, \frac{2}{Y_h}, \dots, \frac{9}{Y_h}$) have been generated before starting the fixed division operation as described in 4.3.3.5. Then the produced partial product vectors are reduced into two vectors represent the intermediate quotient IQ_i^S and IQ_i^{2H}, these two vectors are added to the quotient digit which are produced in the previous clock using CSA reduction tree as the input quotient is represented in two vectors Q_{i-1}^S and Q_{i-1}^{2H} in 4221-BCD format, to avoid carry propagation adders.

The input quotient vectors will be shifted to left three digits as shown in equation 4.5 in order to add the intermediate quotient vectors IQ_i^S and IQ_i^{2H} to achieve correct addition.

$$Q_i = Q_{i-1} \times 10^3 + IQ_{i-1} \tag{4.11}$$

Because there are two vectors IQ_i^S and IQ_i^{2H} represent the intermediate quotient each in 4221-BCD format and these two vectors are used in CSA reduction tree there may be a hidden carry out which can be detected only when adding these two vectors to each other using carry propagation adder as shown in example 4.3, this hidden carry out will increase the value of the intermediate quotient digits with one at the left of the most significant digit, that will affect the produced quotient at this clock. To remove the effect of this hidden one we detect if there is a carry out of the intermediate quotient IQ_i^S and IQ_i^{2H} in each iteration by using generate-propagate technique. Then a negative one vector is produced if there is hidden carry-out and this negative one is subtracted from the intermediate quotient vectors IQ_i^S and IQ_i^{2H}. Numerical example 4.3 illustrates this problem.

Example 4.3: Let $IQ_i^S)_{4221} = E955185847$, $IQ_i^{2H})_{4221} = A681141544$

Which is equivalent to,

$IQ_i^S)_{4221} + IQ_i^{2H})_{4221} = 18FB82A8D8D$, and we will only need 8FB82A8D8D

And the carry out is hidden and this carry-out will affect the correct value of the quotient.

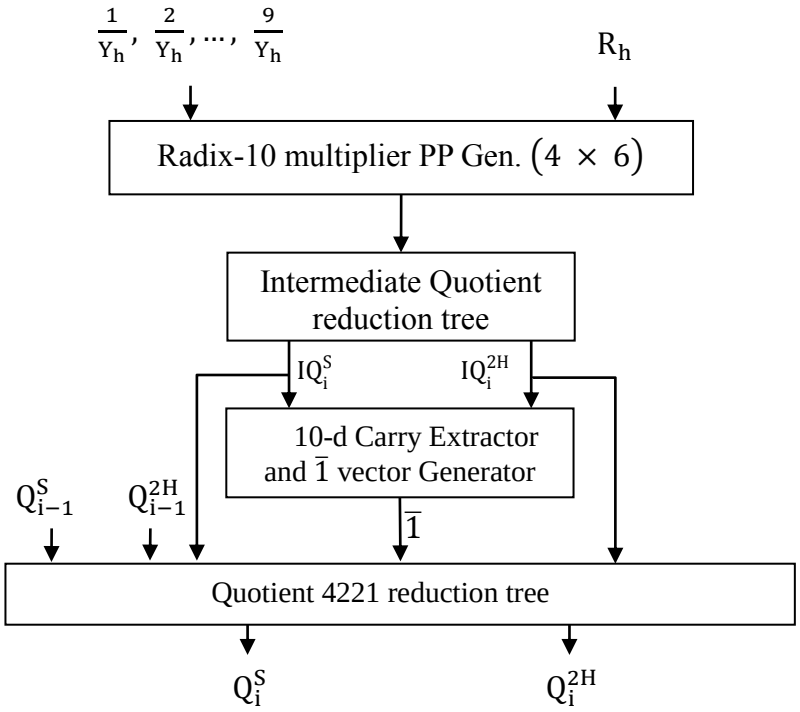

Figure 4.12: Quotient generator module

Hence we will have five vectors which are two vectors for quotient Q_i^S and Q_i^{2H}, intermediate quotient vectors IQ_i^S and IQ_i^{2H} and negative one vector. These five vectors are reduced by using 3:2 CSA reduction tree into two vectors which are the quotient prime Q_i^S and Q_i^{2H} which will be the input for the next clock.

4.2.4 Tail Zeroes Detector

To be compliant with the standard it is required to detect the number of tail zeroes for the final result, as this is very useful in physical measurements to distinguish between the accuracy of measurement. In physical measurements, we distinguish between the cases when the mass of a body is 0.050 kg versus 0.05 kg and say that the first measurement is accurate to the nearest gram while the second is only accurate to the nearest ten grams. So it is very important to generate a result with the same precision of the input.

To do that, we detect the number of tail zeroes in the dividend and the divisor, then we extract the number of two's and five's that are in the dividend and the divisor, then we subtract every extracted divisor element form the corresponding extracted dividend element.

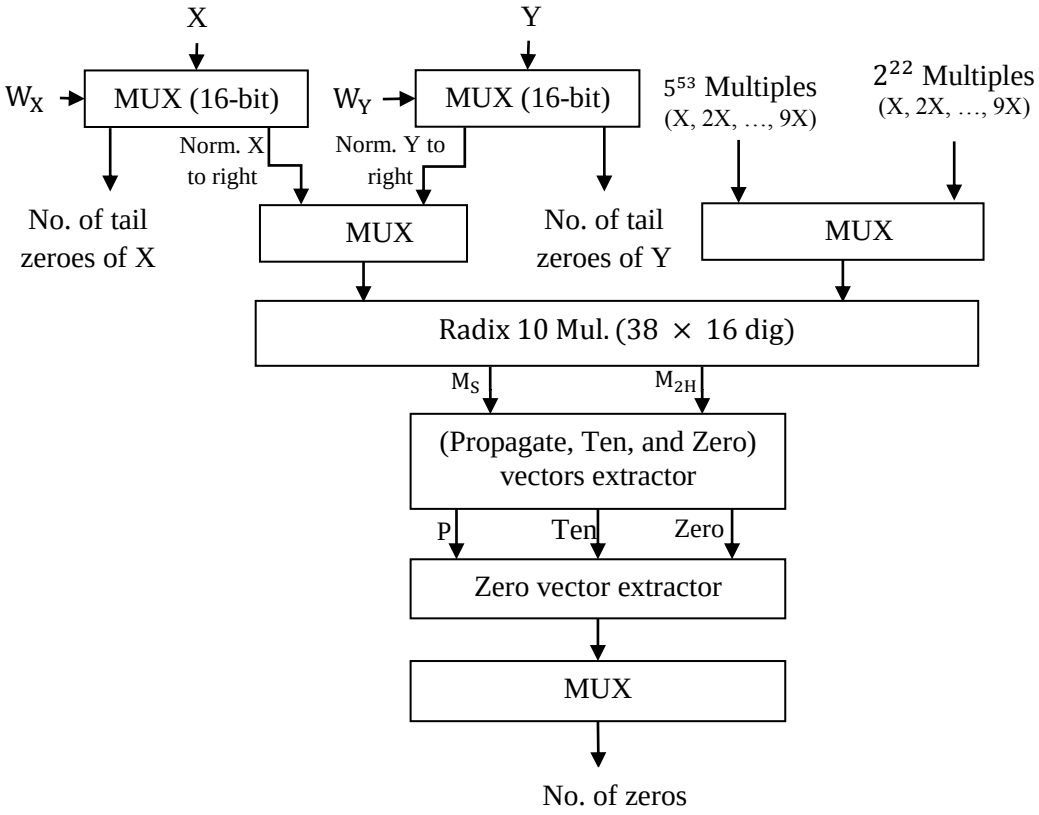

Figure 4.13: Tail zeroes detector

4.2.4.1 Extraction of The Number of Tail Zeroes

Digits X and Y each is shifted to right with values equal to the number of tail zeroes in each digit this done through "Switch CaseX" and cases are chosen according to W_X and W_y for operand X and Y respectively and the shift value represents the number of tail zeroes for both operands. Hence, we will have the number of tail zeroes for dividend X and divisor Y. Then we subtract the number of tail zeroes of the divisor from number of tail zeros of the dividend to extract the difference between them which represents the intermediate tail zeros number. If the intermediate tail zeros number is positive the final result may have tail zeros, but if it is zero or negative the final result will not have tail zeroes. To illustrate this issue table shows numerical examples

X	Y	W_X	W_Y	X_{TZ}	Y_{TZ}	Inter. tail zeros
236000	009470	00..00111000	00..00001110	3	1	$3-1=2$
236000	947000	00..00111000	00..00111000	3	3	$3-3=0$
023600	947000	00..00011100	00..00111000	2	3	$2-3=-1\rightarrow 0$

Table 4.1: Extraction of Intermediate tail zeroes

4.2.4.2 Extraction of The Number of 2's and 5's

The number of tail zeroes of the quotient will be affected by the number of 2's and 5's that are contained by the divisor as if the number of tail zeroes of the result is positive and the divisor contains five or two this will decrement the number of tail zeroes by one, and if these number of 2's and 5's increased the number of tail zeroes will decrease by a value equals to the number of 2's or 5's of the divisor.

But the dividend also may contain tows or 5's, so 2's and 5's of the dividend will cancel the 2's and 5's of the divisor.

We will subtract the number of 2's (5's) of the dividend from the number of 2's (5's) of the divisor and if the difference is positive hence the number of quotient tail zeroes will decrease by a value equals to this result.

To extract number of 2's (5's) in a certain number we multiply normalized to right version of this number by five (two) to the power of maximum number of 2's (5's) using radix 10 multiplier that can be contained by this number for example to find number of 2's (5's) in a decimal number consists of sixteen digit as in our FPD we first extract the maximum number of 2's (5's) can be found in sixteen digit as maximum number of 2's in sixteen digit $= \lfloor \log_2(9999999999999999) \rfloor = 53$ (maximum number of 5's in sixteen digit$= \lfloor \log_5(9999999999999999) \rfloor = 22$) then we multiply normalized to right version of this number by $5^{\text{max.no.of 2's in sixteen digit}}$ ($2^{\text{max.no.of 5's in sixteen digit}}$) which is a fixed value so we will use its multiples directly then we calculate the number of tail zeroes in the multiplication result which represents number of 2's (5's) in this digit.

Multiplication process will result two 54-digit vectors (S and 2H) in 4221 form, and It is required to detect the number of series tail zeros of the multiplication result. So we extract the number of tail zeroes by a new way using "Zero vector extractor" as we aim to generate a 54-bit vector represents the value of the each digit of the multiplication result, if the digit value equals zero, then the corresponding bit will be one, else the corresponding bit will be set to zero.

To avoid CPA we use the following technique,

1- For the multiplication vectors S and 2H we will generate three vectors, the first vector is the propagate vector as each bit will equals one if the corresponding digit equals nine otherwise this bit will equal zero, the second vector is ten vector as each bit of this vector equals one if the corresponding digit equals ten else the bit will equal zero, and the third vector is the zero vector as each bit of this vector equals one if the corresponding digit equals zero else the bit will equal zero.
2- Using these three vector we generate the final zero vector which represents the position of zeroes in the multiplication result vectors according to the following equation

$$\begin{cases} D_{i-1} = 0 \text{ and } D_i = 0 \rightarrow Z_i = 1 \\ D_{i-1} = 0 \text{ and } D_i = 10 \rightarrow Z_i = 1 \\ D_{i-1} = 9 \text{ and } D_i = 9 \rightarrow Z_i = 1 \\ D_{i-1} = 10 \text{ and } D_i = 9 \rightarrow Z_i = 1 \\ \text{o.w} \rightarrow Z_i = 0 \end{cases} \quad (4.12)$$

3- Then the final zero vector is passed through a multiplexer to count the number of tail zeros which represents the number of 2's (5's) in the operand, this done in parallel with multiplication as the multiplication represents the current state and this process represents next state. Figure 4.14 represents numerical example to illustrate the idea.

To extract the no of 2's and 5's for both operands we will need four identical hardware units for each result, so to save the hardware we execute these four processes in loop manner as we use only one hardware unit consists of "$38 \times 16 - \text{dig Multiplier}$" and "no. of tail Zeroes Extractor" and in each iteration we change the input to this module.

S	X	X	X	0	5	8	4	…	2	4	0	0
2H	X	X	X	0	4	2	5	…	7	6	0	0
Sum	X	X	X	0	9	10	9	…	9	10	0	0
P	X	X	X	0	1	0	1	…	1	0	0	0
Ten	X	X	X	0	0	0	0	…	0	1	0	0
Zero	X	X	X	0	0	0	0	…	0	0	1	1
Final Zero	X	X	X	0	1	0	1	…	1	1	1	1

Tail Zeros Series

Figure 4.14: Zero vector generation to obtain number of tail zeros

Now we have "intermediate no. of tail zeroes", "no. of 2's and 5's" of the dividend and the divisor, then we subtract each value of the dividend from its corresponding value of the divisor hence we will have three values which are, Dividend Divisor Tail Zeroes difference, Dividend Divisor Tows difference, and Dividend Divisor 5's difference.

Then we subtract the "Dividend Divisor Tows difference" and "Dividend Divisor 5's difference" from "Dividend Divisor Tail Zeroes difference" if both "Dividend Divisor 2's difference" and "Dividend Divisor 5's difference" are Positive value else Zero is subtracted. Finally number of tail zeroes for the final result has been detected the maximum value allowable for this value is fifteen (4-bits) as shown in figure 4.15. Examples 4.4, 4.5, and 4.6 show numerical example for the extraction of the final result tail zeros value.

Example 4.4:

	W	TZ	Norm. to Right	$\times 2^{22}$	No. of 5's	$\times 5^{53}$	No. of 2's
X = 31250000	0..0011110000	4	3125	XX7200000	5	XX65625	0
Y = 00016680	0..0000011110	1	1668	XX6099072	0	XX12500	2

Then,

$\text{TZ}_{\text{diff}} = \text{X}_{\text{TZ}} - \text{Y}_{\text{TZ}} = 4 - 1 = 3$

$5'\text{s}_{\text{diff}} = \text{Y}_{5'\text{s}} - \text{X}_{5'\text{s}} = 0 - 5 = -5 \rightarrow 0$

$2'\text{s}_{\text{diff}} = \text{Y}_{\text{Tows}} - \text{X}_{\text{Tows}} = 2 - 0 = 2$

$\text{Final result tail zeros} = \text{TZ}_{\text{diff}} - 5'\text{s}_{\text{diff}} - 2'\text{s}_{\text{diff}} = 3 - 0 - 2 = 1$

Example 4.5:

	W	TZ	Norm. to Right	$\times 2^{22}$	No. of 5's	$\times 5^{53}$	No. of 2's
X = 3125000	0..0001111000	3	3125	XX7200000	5	XX65625	0
Y = 0016250	0..0000011110	1	1625	XX5744000	3	XX78125	0

Then,

$TZ_{diff} = X_{TZ} - Y_{TZ} = 3 - 1 = 2$

$5's_{diff} = Y_{5's} - X_{5's} = 3 - 5 = -2 \rightarrow 0$

$2's_{diff} = Y_{Tows} - X_{Tows} = 0 - 0 = 0$

Final result tail zeros $= TZ_{diff} - 5's_{diff} - 2's_{diff} = 2 - 0 - 0 = 2$

Example 4.6:

	W	TZ	Norm. to Right	$\times 2^{22}$	No. of 5's	$\times 5^{53}$	No. of 2's
X = 3124000	0..0001111000	3	3124	XX3005696	0	XX62500	2
Y = 0016250	0..0000011110	1	1625	XX5744000	3	XX78125	0

Then,

$TZ_{diff} = X_{TZ} - Y_{TZ} = 3 - 1 = 2$

$5's_{diff} = Y_{5's} - X_{5's} = 3 - 0 = 3$

$2's_{diff} = Y_{Tows} - X_{Tows} = 0 - 2 = -2 \rightarrow 0$

Final result tail zeros $= TZ_{diff} - 5's_{diff} - 2's_{diff} = 2 - 3 - 0 = -1 \rightarrow 0$

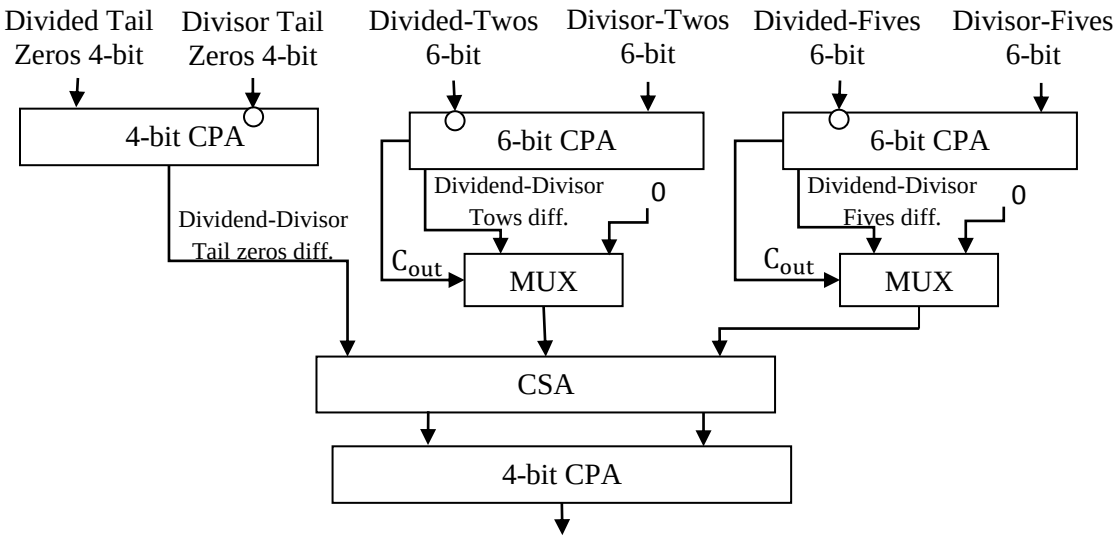

Figure 4.15: Binary processes to find the value of tail zeroes

4.2.5 Final Quotient Preparation and Rounding

Because the dividend is normalized to be $0.1 \leq X < 1$ and the divisor is normalized to be $0.1 \leq Y < 1$, the value of the resulted quotient Q' of the fixed point divider will belong to two intervals, the first one is $1 \leq Q' < 10$ when the normalized dividend is greater than or equal to the normalized divisor, and the other interval is $0.1 \leq Q' < 1$ when the normalized dividend is smaller than the normalized divisor. So it is required to keep the resulted quotient in the range $1 \leq Q' < 10$ to be have the same form of the input operands, to do that the normalized divisor is subtracted from the normalized dividend and if the result is negative which means that the normalized divisor is greater than the normalized dividend hence the quotient will be shifted one digit to left and the exponent will be updated with this shift, otherwise no shift is required.

As mentioned earlier the resulted quotient Q' is an approximation of the infinite correct quotient Q, hence the difference between the finite correct quotient Q up to n+1 digits and the resulted quotient Q' is always,

$$0 \leq Q_{[0:-(n+1)]} - Q'_{[0:-(n+1)]} \leq 10^{-(n+1)} \tag{4.13}$$

Where n represents the desired precision.

Hence, it is required to adjust the resulted quotient Q' to achieve correct rounding so we follow the technique that proposed in [16]. As the resulted quotient Q' is truncated to (n+1) digit then $10^{-(n+1)}$ is added to the truncated result to have Q''. Then the Q'' is multiplied by the divisor Y after that the result is subtracted from the dividend to determine the remainder R as follow,

$$R = X - Q'' \times Y \tag{4.14}$$

This operation is performed by calculating the multiples of Q'' and then the radix-10 (38× 16) multiplier is reused to perform the multiplication $Q'' \times Y$, which results in two vectors each in form 4221-BCD format, these two vectors are subtracted from the dividend using CSA, which produces another two vectors represent the remainder R. the remainder R is checked if it equals zero or not using the same technique that used in 4.3.4.2. To detect the sign of the remainder R we check if the sum of the most significant digit of its vectors equals nine hence negative sign or equals zero hence positive sign.

According to the sign of the remainder R and if it equals zero the final quotient is chosen from Q' and Q'', as if the remainder R is negative value the or zero the chosen quotient Q_{chosen} will be Q'' else the chosen quotient Q_{chosen} will be the most significant n+1 digits of Q' as the most significant n digits represent the desired precision and the least significant digit is the guard digit. The chosen quotient Q_{chosen} will be shifted and rounded to generate the correct output quotient as will be described.

The chosen quotient is called exact quotient if it equals the infinite correct quotient Q, and that can happen when, the normalized dividend is multiple of the normalized divisor. The exact quotient can be detected by that if the remainder R is zero.

The chosen quotient Q_{chosen} will be shifted to right in two cases, the first case when the chosen quotient $Q_{chosen}$is exact. As the exact quotient may contain tail zeroes that can be removed by shifting the quotient to right.

The value of right shift in case of exact quotient will be the difference between the no. of zeros in the quotient vector and the "Final Result Tail Zeroes Value" , but if the " Final Result Tail Zeroes Value" is zero the shift value equals no. of tail zeros in the exact quotient as shown in figure 4.15. The value of shift is added to the intermediate exponent to achieve correct results but if the resulted exponent is greater than the maximum exponent value the quotient will be shifted to right by value equals to the difference between the intermediate exponent and the maximum exponent only and the final exponent is set to the maximum exponent value as shown in figure 4.15, this can be achieved by subtracting the intermediate exponent from the maximum exponent , then we compare the result with " Final Result Tail Zeroes Value " if greater it is ok, else we will use the difference between the intermediate exponent and the maximum exponent the as input to the shifter as shown in figure 4.15.

But if there is an underflow we will use the "underflow value" -which is the difference between the intermediate exponent and the minimum exponent that is generated by exponent module- as the input to the shifter instead of normal result shift value.

For underflow there are four cases,

a. Subnormal number, as the intermediate exponent is lower than the minimum exponent with value below sixteen, the quotient is exact, and the number of the tail zeroes of the exponent is greater than the underflow value. So the quotient is shifted to right with value equals to the underflow value, the exponent is set to minimum exponent value, and the underflow flag will not be raised.
b. Acceptable underflow where, the result exponent is lower than the minimum exponent with a value below sixteen, and the quotient is inexact or may be exact but its tail zeroes number is lower than the underflow value, hence the chosen quotient is shifted the with value equals to the underflow value, set the exponent to the minimum exponent, and the underflow flag is raised.
c. The underflow value equals sixteen where, the difference between the exponent and the minimum exponent value equals sixteen, here we will keep the most significant digit of the chosen quotient in the guard digit position of the final quotient, set all the most significant digits of the final quotient to zero, set the exponent to the minimum exponent, and the underflow flag is raised.
d. Non-acceptable underflow, the difference between the exponent and the minimum exponent value is greater than sixteen, here we set the final quotient to zero, and set the exponent to the minimum exponent.

After determining the required shift value which is k, the chosen quotient Q_{chosen} is shifted to right with this value, the most significant k digits will be set to zero and the shifted out digits will be eliminated. Hence there will be seventeen digits for final quotient Q_{final} as the most significant sixteen digits will be the quotient and least significant digit will be the guard digit to be used in rounding.

For rounding process, if the final quotient is exact and the it shifted to right with value lower than or equals to the tail zeroes value the round value will be zero, but if the quotient is exact and shifted to the right with a value greater than the tail zeroes value or the quotient is

not exact, the round value will be one or zero according to the round mode, the guard digit value, the least significant digit is even or odd, and the sign of the final result. After determining the round value it is added to the most significant sixteen digits of the final quotient to produce the final result.

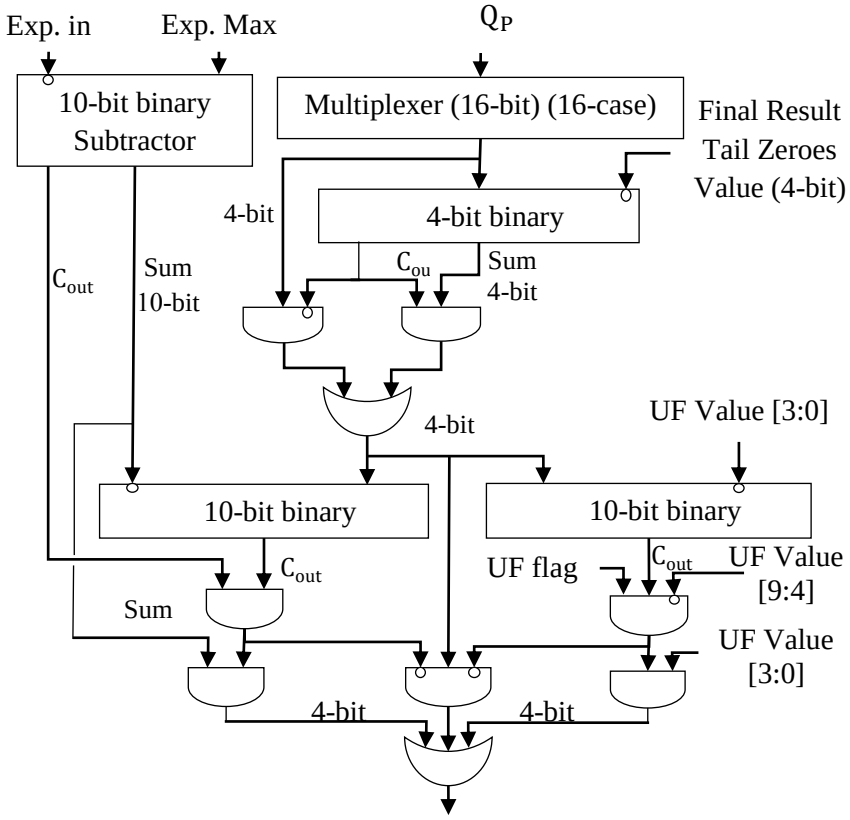

Figure 4.16: Determining the shift value to shift the final quotient

4.2.6 Special Cases

4.2.6.1 Infinity

The final result is infinity if the division operation is one of these cases, the dividend is infinity and the divisor is normal decimal number or zero, the dividend is normal decimal number and the divisor is zero. Also the final result is infinity if there is overflow and it is required to round up (round away from zero, ties up, and so on).

4.2.6.2 Not a number

Not a number special case occurs when, one of the operands is signaling Nan or quite Nan or the division operation is one of this ($\frac{\text{zero}}{\text{zero}}$ or $\frac{\mp\infty}{\mp\infty}$), the result of any invalid operation is always quite Nan.

4.2.7 Flags

4.2.7.1 Overflow

This flag is raised if the intermediate exponent exceeds the maximum exponent (10^{398}, in case of decimal 64). In case of over flow the final result is set to infinity or the maximum representable number according to the round mode and the result sign.

4.2.7.2 Underflow

Under flow is detected when the intermediate exponent is smaller than the minimum exponent (10^{-383}, in case of decimal 64) and the final result is rounded after shifting the final quotient to the right with value equals to the under flow value.

4.2.7.3 Inexact

This flag is raised if the result is rounded as this is detected from the guard digit; also the inexact flag is raised if there is overflow, or underflow.

4.2.7.4 Invalid

The invalid flag is raised in the following cases;

- one of the operands is sNan
- When division operation is $\frac{\text{zero}}{\text{zero}}$ or $\frac{\mp\infty}{\mp\infty}$

4.2.7.5 Zero

The zero flag is raised if the divisor is ± 0 and the dividend is normal number.

4.3 Operation Sequence

The overall operation sequence and all tasks that are performed in each cycle are summarized as follow,

Cycle 1:

Task1 the dividend and divisor operands are converted from DPD format to BCD format.

Task 2 the operands normalization processes are performed by removing the leading zeros of the dividend and the divisor.

Task2, converting the dividend and the divisor from 8421-BCD format to 4221-BCD redundant format.

Cycle 2:

Task1, obtaining the value $\frac{1}{Y_h}$ from a look up table.

Task4, obtaining the multiples of the divisor [2Y, …, 9Y]

Task3, multiplying the dividend by $5^{\text{max.no.of 2's in sixteen digit}}$ to obtain the number of 2's in the dividend in the next cycle.

Cycle 3:

Task1, performing the multiplication process $Y' = Y \times \frac{1}{Y_h}$.

Task2, obtaining the multiples of the divisor high inverse $[2\frac{1}{Y_h}, \ldots, 9\frac{1}{Y_h}]$.

Task3, obtain the number of 2's in the dividend.

Task4, multiplying the dividend by $2^{\text{max.no.of 5's in sixteen digit}}$ to obtain the number of 5's in the dividend in the next cycle.

Cycle 4:

Task1, adding the two vectors of the divisor prime Y'_{M1} and Y'_{M2} which are the result of multiplication process in the previous cycle and calculating the multiples of the divisor prime $[2Y', \ldots, 9Y']$.

Task2, obtain the number of 5's in the dividend.

Task3, multiplying the divisor by $2^{\text{max.no.of 2's in sixteen digit}}$ to obtain the number of 2's in the divisor in the next cycle.

Cycle 5:

Task1, the start of executing AQA fixed point division iterative equations as follow

$$R_i = R_{i-1} - R^h_{i-1} \times Y' \quad (4.15)$$

$$Q_{i-1} = Q_{i-1} \times 10^j + R^h_{i-1} \times \frac{1}{Y_h} \quad (4.16)$$

Task2, obtain the number of 2's in the divisor.

Task3, multiplying the divisor by $2^{\text{max.no.of 5's in sixteen digit}}$ to obtain the number of 5's in the divisor in the next cycle.

Cycle 6:

Task1, the second iteration of the fixed point division operation.

Task2, obtain the number of 5's in the divisor.

Cycle 7 to Cycle 11:

Which are 4 cycles in order to generate 18-digit quotient, we perform the iterative equations, as the partial remainder and quotient inputs are the outputs of the previous equation.

Cycle 12:

Task1, normalized the Quotient Q'

Task2, calculating Q'' and its multiples .

Cycle 13:

Task1, performing the multiplication $Q'' \times Y$

Task2, determining the value which is required to shift the quotient to the right.

Cycle 14:

Task1, checking the status of the remainder R if it equals zero or not and detecting its sign.

Cycle 15:

Task1, The quotient Q_{chosen} is shifted according to the determined value in cycle 13 if the Q_{chosen} is exact or there is underflow.

Task2, the round value is determined according to the round mode, the guard digit, the result sign, and the least significant digit if it is even or odd.

Task3, the round digit which is one or zero is added to the chosen quotient Q_{chosen} to produce the final result.

Task4, the final exponent is calculated by updating the intermediate exponent will the quotient shift value.

Task5, all flags and special signals are generated.

Task6, the output quotient, exponent, and flags of are coded to DPD form to be compliant with the IEEE 754-2008 standard.

Chapter 5. Verification and Synthesis Results

This chapter presents the testing and synthesis results of the proposed Decimal Floating Point Divider.

5.1 Verification

Verifying decimal floating point dividers is a very challenging task, due to the large number of input space as, there are two input operands each is 64-bit length and there are three bits for rounding, so it is impossible to test the DFP unit for all possible cases. However we can test the DFP unit by using large number of test cases which are designed to cover different corners of the normal operation.

The proposed Decimal floating Point Divider was modeled with Verilog, simulated by Modelsim, and verified in Matlab. The decimal unit was verified using 949966 test cases that were proposed by Sayed-Ahmed [43] and can be downloaded from [44], which are specially designed to check the decimal functionality of the unit in many corners. All test cases are passed correctly which give an excellent indication that our decimal division unit is functionally correct.

5.2 Synthesis Results

The proposed DFP design was synthesized using Synopsys design compiler tool working on 65nm low power TSMC65LP kit with 1.2V typical process and 25°C temperature. The latency is measured using FO4 unit which is defined as the delay of an inverter with a fan-out of four inverters. The gate delay of a FO4 inverter is 35 ps, and the area of the smallest NAND2 gate is $1.6\ \mu m^2$. The target of the synthesis was minimizing the cycle time. Synthesis results showed that the cycle time is 1.69 ns (48.28 FO4), and the total area of the design is $250948.8 \mu m^2 (156842\ \text{NAND2})$, where the combinational logic area is $219163.6\ \mu m^2 (78.9\%)$ and the others are registers.

5.2.1 Delay and Area

The critical path is due to Partial Remainder Generation block, which is shown in figure 4.11. Table 5.1 shows the detailed information about the critical path.

Component	latency(ns)	latency (FO4)
Register	0.07	2
Partial Product Gen.	0.47	13.43
Partial Remainder Reduction Tree.	0.67	19.14
BCD Adder	0.48	13.71
Total	1.69	48.28

Table 5.1: Critical path detailed information

The delay and area of some BCD units are shown in table 5.2.

Module	Latency ns	Latency FO4	Area μm^2	Area NAND2
BCD 22-digit Adder	0.48	13.71	8230	5143
BCD Multiplier (38×16 digit)	1.25	35.7	173129.2	108205
Memory block (9000 entries $\times$ 16 bit/entry)	0.55	15.7	42400.8	26500

Table 5.2: Delay and area of BCD units

Table 5.3 shows the comparison of the proposed decimal floating point divider with previously published DFP dividers. For fair comparison, the latency is measured in FO4 unit, and the area consumption is evaluated in terms of smallest NAND2 gate unit.

Divider	Cycle time (FO4)	No. of cycles	latency(FO4)	Area (NAND2)
[25], 2007	25.3	21	531	10,500
[41], 2007	26.8	20	536	13,586
[45], 2006	35.8	19	680	22,600
[24], 2007	13	82	1066	6,600
[42], 2004	12.6	150	1890	-
[9], 2010	138.5	21	2907	-
Proposed DFP Divider	48.2	15	724	156,842

Table 5.3: Comparison of delay and area with others Dividers

From the comparison we conclude that our proposed divider is comparable in terms of latency with dividers [25, 41, and 45] which are based on digit recurrence division algorithm. For Power6 decimal divider [24] and [9] divider which are both based on digit recurrence division algorithm we have better latency. The proposed DFP divider has better latency than Newton-Raphson divider [42] which requires two decimal multiplications every iteration.

In terms of area our divider occupies larger area than dividers [25, 41, 45, and 24], because the 16×38 multiplier which is the main block used in Tail Zeros detector module takes around 70% of the total area as shown in table 5.2. So if calculate the total area without this multiplier the total area will be 48636 NAND2.

Chapter 6. Conclusion and Suggestions for Future Work

This thesis presents new decimal floating point divider, which is based on Accurate Quotient Approximation division algorithm. This algorithm was presented for very high radix binary division technique, and then we modified this algorithm to be suitable for decimal division. To design the decimal floating point divider we design radix-10 multiplier, BCD adder, tail zeroes detector, and many other blocks that can be used for any other decimal arithmetic units.

The functionality of the division unit is verified by around one million test cases that test many corners of the decimal dividers. The divider is synthesized using TSMC65nmLP kit and the results were discussed and compared with previously published arithmetic units.

The proposed DFP divider can be extended to decimal-128 by extending the used multipliers and BCD adders to be suitable for the decimal-128 operands, and increasing the number of the clock cycles to achieve the desired precision.

The functionality of the DFP arithmetic unit can be extended to perform all the decimal floating point operations in the standard.

Our DFP divider takes very large area which is the main drawback of our design, so the future work can focus on decreasing the area and enhance the cycle time.

References

[1] H. Goldstine and A. Goldstine, “The Electronic Numerical Integrator and Computer (ENIAC),” Annals of the History of Computing, IEEE, vol. 18, no. 1, pp. 10 –16, 1996.

[2] D. E. Knuth, “The IBM 650: An Appreciation from the Field,” Annals of the History of Computing, vol. 8, pp. 50 –55, Jan.-Mar. 1986.

[3] A. W. Burks, H. H. Goldstine, and J. von Neumann, “Preliminary discussion of the logical design of an electronic computing instrument,” tech. rep., Institution for Advanced Study, Princeton, 1946.

[4] M. Cowlishaw, “Decimal floating-point: algorism for computers,” in Computer Arithmetic, 2003. Proceedings. 16th IEEE Symposium on, pp. 104 – 111, June 2003.

[5] A. V. Alvarez, “High-performance Decimal Floating-Point Units,” University of Santiago de Compostela, Jan. 2009.

[6] SUN Microsystem, “BigDecimal class, API specification for the Java 2 platform,” 2004. [Online]. Available: http://java.sun.com/j2se/1.3/docs/api/, 2015.

[7] M. F. Cowlishaw “The decNumber C library, version 3.68,” Jan. 2010. [Online]. Available: http://speleotrove.com/decimal/decnumber.pdf, 2015.

[8] Intel Corporation, Intel decimal floating-point math library, 2010. http://software.intel.com/enus/articles/intel-decimal-floating-point-math-library/.

[9] D. Chen, “Algorithms and architectures for decimal transcendental function computation,” University of Saskatchewan, January, 2010.

[10] F. Busaba, C. Krygowski, W. Li, E. Schwarz, and S. Carlough, “The IBM z900 decimal arithmetic unit,” in Signals, Systems and Computers, 2001. Conferenc.

[11] IEEE, Inc., IEEE Standard for Binary Floating-Point Arithmetic, Mar. 1985.

[12] IEEE Standard for Radix-Independent Floating-Point Arithmetic, Mar. 1987.

[13] IEEE, Inc., IEEE 754-2008 standard for floating-point arithmetic, Aug. 2008.

[14] M. Cowlishaw "Densely packed decimal encoding," Computers and Digital Techniques, IEE Proceedings -, vol. 149, pp. 102 –104, May 2002.

[15] Wikipedia, "Binary Integer Decimal," Available: https://en.wikipedia.org/wiki/Binary_Integer_Decimal, 2015.

[16] J. Thompson, N. Karra, and M. J. Schulte, "A 64-bit decimal floating point adder," in the 3rd IEEE Computer society Annual Symposium on VLSI (ISVLSI-3), pp. 297 –298, Feb. 2004.

[17] L. K. Wang and M. J. Schulte, "Decimal floating-point adder and multifunction unit with injection-based rounding," in the 18th IEEE Symposium on Computer Arithmetic (ARITH-18), pp. 56 –68, June 2007.

[18] L. K. Wang and M. J. Schulte, "A decimal floating-point adder with decoded operands and a decimal leading-zero anticipator," in the 19th IEEE Symposium on Computer Arithmetic (ARITH-19), pp. 125 –134, June 2009.

[19] A. Vazquez and E. Antelo, "Conditional speculative decimal addition," in the 7th Conference of Real Numbers Computers (RNC-7), pp. 47 – 57, July 2006.

[20] H. A. H. Fahmy, R. Raafat, A. M. Abdel-Majeed, R. Samy, T. ElDeeb, and Y. Farouk, "Energy and delay improvement via decimal floating point units," in the 19th IEEE Symposium on Computer Arithmetic.

[21] M. A. Erle, M. J. Schulte, and B. J. Hickmann, "Decimal floating-point multiplication via carry-save addition," in the 18th IEEE Symposium on Computer Arithmetic (ARITH-18), pp. 46 –55, June 2007.

[22] B. Hickmann, A. Krioukov, M. Schulte, and M. Erle, "A parallel IEEE p754 decimal floating-point multiplier," in the 25th IEEE International Conference on Computer Design (ICCD-25), pp. 296 –303, Oct. 2007.

[23] A. Vazquez, E. Antelo, and P. Montuschi “Improved design of high-performance parallel decimal multipliers,” IEEE Transactions on Computers, vol. 59, pp. 679 – 693, May 2010. (ARITH-19), pp. 221 –224, June 2009.

[24] E. M. Schwarz and S. R. Carlough, “Power6 decimal divide,” in the 18th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP-18), pp. 128 –133, July 2007.

[25] A. Vazquez, E. Antelo, and P. Montuschi, “A radix-10 SRT divider based on alternative bcd codings,” in the 25th IEEE International Conference on Computer Design (ICCD-25), pp. 280 –287, Oct. 2007.

[26] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O’Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden, “IBM POWER6 microarchitecture,” IBM Journal of Research and Development, vol. 51, pp. 639 –662, Nov. 2007.

[27] J. Friedrich, B. McCredie, N. James, B. Huott, B. Curran, E. Fluhr, G. Mittal, E. Chan, Y. Chan, D. Plass, S. Chu, H. Le, L. Clark, J. Ripley, S. Taylor, J. Dilullo, and M. Lanzerotti, “Design of the Power6 microprocessor,” in the 54th IEEE International Conference on SolidState Circuits (ISSCC-54), pp. 96 –97, Feb. 2007.

[28] D.Piso, J.A.Piñeiro, J.D.Bruguera, "Analysis of the Impact of Different Methods for Division/Square Root Computation in the Performance of a Superscalar Micro-processor", Proceedings of EUROMICRO Symposium on Digital System Design (DSD 2002), Dordmunt (Germany), 2002.

[29] S.F. Oberman. "Design Issues in High Performance Floating Point Arithmetic Units". PhD thesis, Stanford University, January 1997.

[30] R.E. Bryant. Bit-Level Analysis of an SRT Divider Circuit. In Proceedings of the 33rd Annual Conference on Design Automation, Pages 661-665, Las Vegas, NV, USA, June 1996. ACM Press.

[31] C.B. Moler. Atale of Two Numbers. SIAM News, 28(1): 16-16 January 1995.

[32] S.F Oberman and M.J.Flynn. Division Algorithms and Implementations. IEEE Transactions on Computers, 48(6):833-854, August 1997.

[33] Ercegovac and Tomas Lang. Division and square-root: digit-recurrence algorithms and implementations. Kluwer Academic Publishers Group, Norwell, MA, USA, and Dordrecht, the Netherlands, 1994. ISBN 0-7923-9438-0. x + 230 pp. LCCN QA76.9.C62 E73 1994.

[34] I. Koren, Computer Arithmetic Algorithms. A K Peters,Ltd., 2002.

[35] J. E. Robertson. A new class of digital division methods. IRE Transactions on Electronic Computers, EC-7(3):88{92, September 1958. CODEN IRELAO. ISSN 0367-9950.

[36] John L. Hennessy and David A. Patterson. Computer Architecture "A Quantitative Approach". Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1990. ISBN 1-55860-069-8. Xxviii + 594 pp. LCCN QA76.9.A73 P377 1990.

[37] Norman R. Scott. Computer Number Systems and Arithmetic. Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1985. ISBN 0-13-164211-1.

[38] R. E. Goldschmidt. Applications of division by convergence. Thesis (M.S.), Dept. of Electrical Engineering, MIT, Cambridge, MA, USA, June 1964. 44 (est.) pp.

[39] D. Wong and M. Flynn, "Fast Division Using Accurate Quotient Approximations to Reduce the Number of Iterations," IEEE Trans. Computers, vol. 41, no. 8, pp. 981-995, Aug. 1992.

[40] W. S. Briggs, T. B. Brightman, and D.W.Matula. Method and Apparatus For Performing Division Function Using A Rectangular Aspect Ratio Multiplier. US Patent 5 046 038, US Patent Office, September 1991.

[41] T. Lang and A. Nannarelli, "A Radix-10 Digit-Recurrence Division Unit: Algorithm and Architecture," Computers, IEEE Transactions on, vol. 56, pp. 727–739, June 2007.

[42] L.-K. Wang and M. Schulte, "Decimal floating-point division using Newton-Raphson iteration," in Application-Specic Systems, Architectures and Processors, 2004. Proceedings. 15th IEEE International Conference on, pp. 84 – 95, Sept. 2004.

[43] A. Sayed-Ahmed, H. Fahmy, and M. Hassan, "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, pp. 1153–1157, 2010.

[44] "Cairo University Test vectors, Available Online At: http://eece.cu.edu.eg/hfahmy/arith_debug/index.htm, 2015.

[45] H. Nikmehr, B. Phillips, and C.-C. Lim. Fast decimal floating-point division. IEEE Trans. VLSI Systems, 14(9):951–961, 2006.

دائرة قسمة للأعداد العشرية ذات النقطة العائمة مطابقة للمواصفة المعيارية IEEE 754-2008

إعداد

أحمد حمدى أحمد خليل

رسالة مقدمة إلى كلية الهندسة ، جامعة القاهرة

كجزء من متطلبات الحصول على درجة ماجستير العلوم

في هندسة الالكترونيات والاتصالات الكهربية

كلية الهندسة ، جامعة القاهرة

الجيزة ، جمهورية مصر العربية

2015

58

دائرة قسمة للأعداد العشرية ذات النقطة العائمة مطابقة للمواصفة المعيارية IEEE 754-2008

إعداد

أحمد حمدى أحمد خليل

رسالة مقدمة إلى كلية الهندسة ، جامعة القاهرة

كجزء من متطلبات الحصول على درجة ماجستير العلوم

في هندسة الالكترونيات والاتصالات الكهربية

تحت إشراف

أ.د. حسام على حسن فهمى

كلية الهندسة ، جامعة القاهرة

الجيزة ، جمهورية مصر العربية

2015

دائرة قسمة للأعداد العشرية ذات النقطة العائمة مطابقة للمواصفة المعيارية IEEE 754-2008

إعداد

أحمد حمدى أحمد خليل

رسالة مقدمة إلى كلية الهندسة ، جامعة القاهرة

كجزء من متطلبات الحصول على درجة ماجستير العلوم

في هندسة الالكترونيات والاتصالات الكهربية

يعتمد من لجنة الممتحنين:

ا.د. **حسام على حسن فهمى** المشرف الرئيسي

أ.م.د. **أحمد نادر محى الدين**

ا.د. **السيد مصطفى سعد**

(أستاذ الدوائر الإلكترونية، كلية الهندسة، جامعة حلوان)

كلية الهندسة ، جامعة القاهرة

الجيزة ، جمهورية مصر العربية

2015

دائرة قسمة للأعداد العشرية ذات النقطة العائمة مطابقة للمواصفة المعيارية IEEE 754-2008

على الرغم من ان النظام الثنائى لتمثيل الأرقام هو الامثل لأجهزة الحاسوب الكهربية ولكن ان النظام العشرى هو النظام الأكثر استخداما لتمثيل الأرقام وأيضا فى العمليات الحسابية فى حياة البشر وبخاصة فى التطبيقات المالية. ولكن لا يستطيع النظام الثنائى تمثيل جميع الأرقام العشرية بدقة لذلك اشتملت المواصفات المعيارية IEEE 754–2008 على النظام العشرى لتمثيل الأرقام ذات النقطة العائمة وايضا تعريف العمليات الحسابية. تعتبر عملية القسمة واحدة أساسية من هذة العمليات حيث يتطلب قسمة عدد عشرى على عدد اخر عشرى كل منهما ذو نقطة عائمة.

هذه الرسالة تقدم تصميم وتنفيذ لقاسم جديد مبنى على حل حسابى يسمى "ناتج قسمة دقيق مقرب" " Accurate Quotient Approximation" والذى ينطوى تحت عملية قسمة ذات اسس كبيرة. هذا التصميم تكرارى يعتمد على تحسين ناتج القسمة فى كل مرة عن المرة السابقة. يستخدم مقلوب مقرب من المقسوم عليه يضرب فى الجزء الأعلى من الباقى من المرحلة السابقة لإخراج ناتج قسمة وسطى يضاف إلى ناتج القسمة من المرحلة السابقة لإخراج ناتج قسمة جديد، فى البداية يكون الباقى هو المقسوم، لكى يتم إستخراج الباقى الجديد فى كل مرحلة يتم ضرب الجزء الأعلى من الباقى القديم فى مقلوب مقرب من المقسوم عليه فى المقسوم عليه وهكذا تكررهذه العملية حتى الوصول إلى الدقة المطلوبة لناتج القسمة. هذا القاسم يمكنه إخراج ثلاثة أرقام عشرية فى المرة الواحدة. هذا القاسم مطابق للمواصفة المعيارية IEEE 754–2008. التصميم يدعم توجهات التقريب الخمسة فى المواصفات المعيارية IEEE 754– 2008 فضلا عن إتجاهين للتقريب اخرين يستخدمان على نطاق واسع. بعد ذلك تم تمديد هذا القاسم ليقوم بتنفيذ عمليات القسمة على الأرقام ذات نقطة عائمة.