A REDUNDANT DIGIT FLOATING POINT SYSTEM

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Hossam A. H. Fahmy
June 2003

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Michael Flynn
(Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Martin Morf

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

David Dill

Approved for the University Committee on Graduate Studies.

Abstract

Arithmetic operations are among the most basic instructions in microprocessors, digital signal processors and graphics accelerators. Addition is the most frequent arithmetic operation in numerically intensive applications. Multiplication follows closely and then division and other elementary functions. The speed of those arithmetic operations is also often directly linked to the overall performance of the computers. The work presented in this thesis proposes several techniques to improve the effectiveness of floating point arithmetic units.

A partially redundant number system is used as an internal format for floating point arithmetic operations. The redundant number system is based on signed digits and enables carry free arithmetic operations to improve the performance. Conversion from the proposed internal format back to the standard IEEE format is done only when an operand is written to memory. A detailed discussion of an adder and a multiplier using the proposed format is presented and the specific challenges of the designs are explained. Beside the redundancy, the proposed units include further enhancements that increase the floating point performance such as a hexadecimal based number format and a postponed rounding technique.

A time delay model is developed and applied to analytically predict the performance of the floating point units. The predicted delays are then compared to state-of-the-art designs. The comparison is done over a range of operand widths, fan-in and radices to show the merits of each implementation. The proposed system achieves better performance for double precision and larger operand width. Transistor simulation of the complete adder and multiplier confirm the performance advantage predicted by the analytical model. A brief description of a divider using the proposed format is also presented.

The proposed internal format and arithmetic units comply with all the rounding modes of the IEEE 754 floating point standard.

Acknowledgments

أَنِ ٱشْكُرْ لِى وَلِوَالِدَيْكَ

Thank Me and your parents

ٱلْحَمْدُ لِلَّهِ ٱلَّذِى هَدَانَا لِـهَذَا وَمَا كُنَّا لِنَهْتَدِىَ لَوْلَا أَنْ هَدَانَا ٱللَّهُ

All thanks and gratitude to God who guided us to this and we were not to be guided if it were not for Him guiding us.

I wish to take a moment for myself and for any reader of my humble thesis to think about all the bounties that God has given us all and to thank Him for it. To specifically speak about my work towards the PhD, the least I can say is that it went through multiple joyful and sorrowful moments. Anyone doing a PhD passes through such times when the results are great and other moments when the results indicate that the proposed ideas were wrong. In all those moments God was helping and guiding me and I am thankful for that.

As instructed by my tradition and as life has taught me, parents are always a resource for counseling and help. I cannot thank them enough for what they did for me since my birth. May *Allah* shower them in His mercy on the day of judgment.

I had the good fortune of being a student of three great men with whom I worked closely at Stanford: Prof. James Harris, Prof. Martin Morf. and Prof. Michael Flynn.

In James Harris I saw a practical example of putting the seed in the ground and watering it to grow even if another person is going to use the shade and fruit of the tree and not you. I even learned that this attitude of generosity and helping others is how life should be. James Harris was my advisor for the first two years when I was working with his colleague Richard Kiehl. I learned much from Prof. Kiehl but unfortunately he had to leave Stanford. Prof. Harris, or "the coach" as he likes us to call him, supported me financially and morally even after Prof. Kiehl left. Thank you Coach and I hope for you the best in coaching generations of students to come.

Martin Morf is a physical realization of an "internet." He has links to a large number of research groups at Stanford and personally worked in so many and varied fields. I had wonderful and long discussions with him that taught me about the need to be comprehensive and aware of other aspects of knowledge outside of my narrow field of research. We could speak in one session about topics

covering quantum theory, Divine laws, biology, politics, mathematics, ... and in all of them he would be quoting references and giving information. He was the one who introduced me to Prof. Flynn when I was looking for a new research group to join for continuing my PhD. Prof. Morf, I did not manage in my research to formulate a theory that unifies all the things we discussed but I hope that you and I can continue our search for the truth that unifies the universe.

If you are looking for maturity and flexibility in a person then Prof. Flynn is someone you ought to meet. The general student population at Stanford know him as a great teacher and a great researcher but within his research group we know more: he is, on top of that, a great advisor. His style of working with the students is accommodating. He gives them all the freedom they need to grow in their research and in their personality as well. His long experience provides him with wisdom that added beauty to his refined character. He was quite generous with his advice when I needed it and he supervised my work with parental love. He reminded me of great rivers which are flexible in turning graciously around mountains in their ways and provide their clear waters for the seekers. Thank you Prof. Flynn for being what you are. I hope that you will continue forever being a great river with clear waters for your students, turning away from the problems of life and continuing in the path, nourishing all that surrounds.

The amount of support of my wife and the rest of my family during the years of research is something that only God knows. With no words available to show enough gratitude I say: "may *Allah* reward you the best"

May *Allah* bless and reward all those who helped me and bless you too, gentle reader who is about to read this document. If you benefit from it, please pray for me.

Contents

Abstract **iv**

Acknowledgments **v**

1 Introduction **1**
1.1 Meaning of some crucial words 1
1.2 Benefits and objectives 3
1.3 Reviewing floating point standards 3
1.4 Floating point operations 7
1.5 Layout of the research 8

2 Redundant alternatives **9**
2.1 Why an alternative format 9
2.1.1 Residue Number System 10
2.1.2 Redundant Representations 11
2.2 Overview of the proposed system 13
2.2.1 Reasons behind the choice of field widths and exponent base 14
2.2.2 Conversion issues 18
2.3 Potential gains from the proposal 20

3 Modeling reality **22**
3.1 What is "reality", why do we model it and why do we use parameters? 22
3.2 The time delay model and its validation 24
3.3 Levels of evaluation 26

4 Addition unit **28**
4.1 Conventional adder designs and their time delays 28
4.2 Proposed adder design 30
4.2.1 First challenge: The leading digit detection 33

4.2.2 Second challenge: Rounding . . . 35
4.3 Adder delay analysis and comparisons . . . 36
4.3.1 Conventional systems . . . 38
4.3.2 Comparison results . . . 41
4.4 Simulation results . . . 44
4.5 Adder conclusions . . . 45

5 Multiplication unit **46**
5.1 Conventional Multiplier designs and their time delays . . . 46
5.2 Proposed Multiplier design . . . 47
5.2.1 First challenge: "Negative" bits and Booth recoding . . . 48
5.2.2 Second challenge: When to round . . . 56
5.3 Multiplier delay analysis and comparisons . . . 58
5.3.1 Conventional systems . . . 60
5.3.2 Comparison results . . . 61
5.4 Simulation results . . . 61
5.5 Multiplier conclusions . . . 64

6 Division and elementary functions unit **65**
6.1 Conventional Divider designs and their time delays . . . 65
6.2 Conventional elementary functions units and their time delays . . . 67
6.3 Proposed Divider and elementary functions unit . . . 67
6.4 Delay analysis and comparison . . . 69
6.5 Divider and elementary functions unit conclusions . . . 71

7 Putting it all together **73**
7.1 Speed impact of the overall system . . . 73
7.2 Area impact of the overall system . . . 75
7.3 Conclusions from a system perspective . . . 78

8 Conclusions and future work **79**

A Leading digit detection **82**

B Rounding logic **94**
B.1 Rounding in the adder . . . 94
B.2 Rounding in the multiplier . . . 98
B.3 Rounding conclusions . . . 104

C Implementation details of the adder **105**
C.1 SD adder block 105
C.2 Details of the floating point adder 109
C.3 Conclusions 121

D Implementation details of the multiplier **125**
D.1 Basic blocks 125
D.2 Details of the floating point multiplier 127
D.3 Conclusions 139

Bibliography **140**

List of Tables

1.1 Maximum and minimum exponents in the single and double IEEE precisions. 5
1.2 Encodings of the special values and their meanings. 5
1.3 Frequency of floating point instructions on the MIPS architecture for five programs of the benchmark SPECfp2000. 7

3.1 Time delay of various components in terms of number of $FO4$ delays. f is the maximum fan-in of a gate and n is the number of inputs. 26

4.1 Rounding value for the four IEEE modes and different fractional ranges 36
4.2 Effect of r on the relative improvement of the adder for $n = 80$. 42
4.3 Circuit statistics and simulation results for the floating point adder. 45

5.1 Biased Booth 2. 50
5.2 Booth 2 recoding with the extra bit. 53
5.3 Booth 3 recoding with the extra bit. 54
5.4 Effect of r on the relative improvement of the multiplier for $n = 80$. 61
5.5 Circuit statistics and simulation results for the floating point multiplier. 64

6.1 Effect of r on the relative improvement of the division and elementary functions unit for $n = 80$. 71

7.1 Time delay and width increase at $n = 80$ and $f = 3$. 78

B.1 Logic equations for r_p and r_n 96
B.2 Logic equations for the rounded LSD 97

C.1 Enable values for the Sticky digit calculation. 120

List of Figures

1.1 IEEE single and double floating point number format. 4

2.1 The proposed SD format for floating point numbers. 14

3.1 A chain of $FO4$ inverters. 24

4.1 The conventional two-path adder. 29
4.2 Block diagram of the two-path adder. 31
4.3 Time delay versus significand width for different fan-in values. 43
4.4 Comparison between $r = 4$ and $r = 8$ at $f = 3, 4$. 44

5.1 Conceptual block diagram of the multiplier. 52
5.2 The building block of the Booth 2 recoding. 57
5.3 Producing the negative of one digit. 59
5.4 Multiplier time delay versus significand width for different f and r values. 62

6.1 Division and elementary functions unit [1, 2]. 68
6.2 Time delay versus significand width for different f and r values. 72

7.1 Relative speed improvement versus significand width. 76
7.2 Width of the floating point number versus the significand width. 77
7.3 Relative increase in the width of the proposed format. 78

A.1 P-recoding implementation. 83
A.2 Encoding the position of the leading digit with a hierarchical tree approach. 90
A.3 Leading digit detection with a hierarchical tree approach. 91
A.4 Encoding the position of the leading digit with a priority encoder. 91
A.5 Production of the n_i and z_i signals. 92

B.1 Rounding of the least significant digit. 95
B.2 Encoding the signed sticky digit using a priority encoder. 99

B.3 Multiplying one operand by the rounding part of the other. 101
B.4 Multiplying the rounding parts of the two operands. 103

C.1 Calculation of the possible outcomes for one digit addition. 107
C.2 Three consecutive digits in a signed digit adder. 108
C.3 General view of the floating point adder. 110
C.4 Cancellation path of the floating point adder. 111
C.5 The adder of the cancellation path. 112
C.6 Subtracting the carries in the cancellation adder. 113
C.7 Subtracting the operands in the cancellation adder. 114
C.8 Multiplexer choosing between the far and cancellation paths. 115
C.9 The far path of the floating point adder. 117
C.10 Rounding blocks in the far path. 118
C.11 Logic for checking the complete shift of the lesser operand. The lower part checks if the exponent difference is less than two. 120
C.12 Adder of the far path. 122
C.13 Evaluation of the guard, round and sticky digits in the far path. 123
C.14 N-recoding in the most significant part of the far path adder. 124

D.1 Schematic of the floating point multiplier. 126
D.2 The $[4:2]$ compressor. 126
D.3 A 58 bits row of $[4:2]$ compressors. 127
D.4 Generation and compression of 4 partial products. 128
D.5 Partial product multiplexer. 129
D.6 The four to one multiplexer with default equal to zero. 129
D.7 The preparation of mX. 130
D.8 Correction for the use of mX in case of an LSD of -16. 131
D.9 Inclusion of the correction for mX and the two special rounding PPs. 131
D.10 The complete partial product tree. 132
D.11 The final adder of the multiplier. 134
D.12 One digit of the final adder of the multiplier. 135
D.13 The N-recoding in the most significant digits of the final adder. 136
D.14 Evaluation of the guard, round and sticky digits. 137
D.15 The special adder for the sticky digit calculation. 138

Chapter 1

Introduction

1.1 Meaning of some crucial words

This thesis is titled "A Redundant Digit Floating Point System" and an explanation of this name is needed to introduce the subject. From a linguistic point of view, the word *redundant* means exceeding what is necessary or normal. By *digit*, we mean the numerals 0 to 9 in the conventional decimal system or one of the elements that combine to form numbers in a system other than the decimal system. The decimal system is also called base-10 system and its digits range from 0 to 9, i.e. from 0 to $10-1$. For the decimal system, 10 is called the radix and the digits usually go up to the radix minus one. The same idea applies for other systems. For example, in a binary (base-2) system the digits usually are 0 or 1, in a base-8 system the digits are usually 0 to 7. A number X with n digits ($x_{n-1}, \cdots, x_0$ in the radix β) can be written as $x_{n-1}\, x_{n-2}\, x_{n-3}\, \cdots x_1\, x_0$. The x_0 is taken to represent the units or β^0 values, the x_1 represents the β^1 values, the x_2 represents the β^2 values and so on. The total value of X is given by $X = \sum_{i=0}^{i=n-1} x_i \beta^i$. Such a system is called a weighted positional number system since each position has a weight and the digits are multiplied by that weight. This system was invented in India and developed by the Muslims who called it hisab al-hind حِسَابُ الهِندِ [3] or Indian reckoning in English.

That Indo-Arabic system (also known as the Muslim system) was later introduced to Europe and replaced the Roman numerals. That is the reason why the numerals 0 to 9 are known in the west as the Arabic numerals. A simple idea links the Roman system to the much older Egyptian system: the units have a symbol used to count them and that symbol is repeated to count for more than one. A group of five units has a different symbol. Ten units have another symbol, fifty units have yet another symbol and so on. This Roman system only survives today for special applications like numbering the chapters of a book but is not in much use in arithmetic. Another number system that existed in history is the Babylonian system which was a sexadecimal system and it survives today in the way we tell the time by dividing the hour into sixty minutes and the minute into sixty

seconds. In the work presented here, with the exception of a comparison in Chapter 2 with a system based on the older Chinese number system, we will consider only the weighted positional system.

In the examples given above for the weighted positional system, we assumed that the number of numerals or symbols used is exactly equal to the radix of the system and that no redundancy is available. One form of redundancy is to introduce more numerals so that the total number of numerals exceeds the radix. As such, multiple representations of a given value become possible. This is the type of redundancy used throughout this thesis unless otherwise noted. Other types of redundancy might include the use of additional information to help in error detection and correction for example. We will not concern ourselves for the time being with those other forms of redundancy until chapter 8.

As for the *"floating point"* part of the thesis title, the word *floating* is used here with the meaning "continually drifting or changing position" and the *point* to which we refer is the fractional point delimiting the integer part of a number from its fractional part. The reason this point is floating is that a normalized scientific notation is used. In such a notation, a number can be represented by one non-zero digit preceding the fractional point and the subsequent digits following the point multiplied by the radix of the system raised to some exponent ($x_{n-1}.x_{n-2}\cdots x_0 \times \beta^{exp}$, with $x_{n-1} \neq 0$.) An alternative definition is to say that the number is represented by the digit zero preceding the fractional point and then a fractional part starting with a non-zero digit and all of that multiplied by the radix raised to some exponent ($0.x_{n-1}x_{n-2}\cdots x_0 \times \beta^{exp}$, with $x_{n-1} \neq 0$.). For example, in decimal, the number three hundred and forty two can be represented as 3.42×10^2 according to the first definition and as 0.342×10^3 according to the second definition. Using the first definition, if we multiply 3.42×10^2 by 10 the result of 34.2×10^2 must be normalized to 3.42×10^3. The fractional point changed its position after this multiplication, hence the name floating point. Obviously, to represent a negative number an additional piece of information can be added to the representation to define the sign. In both definitions mentioned above, the number zero cannot be correctly represented as a *normalized* number since it does not have any non-zero digits and needs a special treatment. In this thesis, we will adopt the first definition with one non-zero digit preceding the floating point to denote a normalized number. The exact definition of normalized numbers using the redundant digits is formalized in chapter 2.

The word *system* is generally used in English to denote "a regularly interacting or interdependent group of items forming a unified whole." So, the work presented here is not only going to treat floating point numbers but also the use of redundant digits to form a complete system performing addition, subtraction, multiplication and division.

Such a system is not a pure mathematical system. It is rather for use in computers. This adds some constraints but the main one is the limit on the number of digits to be used. This limitation translates into the representation of only a finite set of numbers. All other numbers from the set of real numbers are not representable. Some of the effects of this finitude are clear. Definitely any

irrational number with an infinite number of digits after the fractional point is not representable. The same case applies for rational numbers whose representation as a floating point number is beyond the number of digits available. For example, if we assume a decimal number system with five digits after the fractional point then a number like $1234567/500000 = 2.469134$ cannot be represented exactly. Increasing the number of digits used to six may help to include an accurate representation for that rational number, however, numbers like $\sqrt{2}$, e and π are still not represented. This finitude also means that there is an upper bound on the numbers that are representable. If an arithmetic operation has a result beyond this upper limit a condition called overflow occurs and either the hardware or the software running on top of it must handle the situation differently to get a meaningful result. Similarly, a lower bound on the minimum absolute value of fraction exist and a condition called underflow occurs if an arithmetic operation has a result below this limit.

1.2 Benefits and objectives

This work investigates the question of when does such a redundant digit floating point system outperform the conventionally used systems in their speed of operation. The emphasis is on speed because the design requirements for a Floating Point Unit (FPU) adder or a multiplier are usually a specified minimum frequency of operation or a maximum time delay. Sometimes both, the frequency and the delay, are optimized as these are different parameters and not reciprocals. In non-pipelined[1] hardware they are reciprocals while in pipelined FPUs the frequency is determined by inverse of the time delay of one stage of the pipeline while the total time delay is approximately the single stage time delay multiplied by the number of stages. So frequency and total time delay are related but not necessarily reciprocals.

The design constraints on the other hand are the area and the power budget alloted to the FPU. With larger chips currently produced, area is not as critical a constraint as power. This is even more true for the battery operated portable devices. We briefly consider the subject of power consumption in chapter 8. However, the main objective of this research is the quest for more speed.

1.3 Reviewing floating point standards

If we want to use a floating point system, we need to define several aspects. Those include the number system, the location of the fractional point and whether the numbers are normalized or not. Because of this need, several formats arose, some were *de facto* standards used by large companies[2] and some were developed by standardization bodies such as the Institute of Electrical

[1] pipelined hardware is where the circuit performing the operation is divided into stages. Each stage gets its inputs from storage elements that saved the output of the previous stage. A clock synchronizes the whole circuit.

[2] Some of those are the floating point system of IBM and Cray.

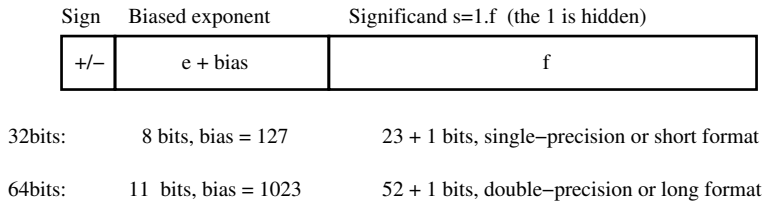


Figure 1.1: IEEE single and double floating point number format.

and Electronics Engineers (IEEE). The IEEE standard was developed in order to support portability between computers from different manufacturers as well as different programming languages. It emphasizes issues such as rounding the numbers correctly to get reliable answers. At the time of this writing, the IEEE standard is the most widely used in general purpose computation. Hence it will be explained in more detail.

The IEEE produced a standard for binary floating point arithmetic in 1985 [4] and a second complementary one for radix independent floating point arithmetic in 1987 [5]. Both standards propose two precisions for the numbers: a single precision and a double precision. The single and double precision numbers in the binary IEEE standard are formed as shown in Fig. 1.1. The most significant bit is the sign bit ($sign$) which indicates a negative number if it is set to 1. The following field denotes the exponent (e) with a constant bias added to it. This excess bias is a positive number added to the exponent field to enable the representation of negative exponents easily as positive binary numbers. So, a real exponent of -3 is represented as a biased exponent of 124 when the bias is 127. With this biased exponent notation, as numbers get smaller and have negative exponents they gradually approach the value of zero which is represented by an all zeros bit string. As shown Fig. 1.1, the remaining part of the number is normalized to have one non-zero bit to the left of the floating point. This last part is called the significand because it is not strictly a fraction but consists of an integer portion and a fractional portion. Since this is a non-redundant binary system, any bit is either 0 or 1. Hence, the normalized numbers must have a bit of value 1 to the left of the floating point. The value of the bit is always known and thus there is no need to store it and it is implied. This implicit bit is called the 'hidden 1.' Only the fractional part (f) of the significand is then stored in the standard format. To sum up, the number given by the standard format has the value $(-1)^{sign} \times 2^e \times 1.f$.

The biased exponent has two values reserved for special uses: the all ones and the all zeros. For the single precision those values are 255 and 0 giving a maximum allowed real exponent (E_{max}) of $254 - 127 = 127$ and a minimum exponent (E_{min}) of -126. Table 1.1 summarizes the maximum and minimum exponents for the single and double precision. As for the special values, their interpretation is as shown in Table 1.2. If the exponent field is all ones and the fraction field is not zero then this represents what is called 'Not a Number' or NaN in the standard. This is a symbolic entity

Table 1.1: Maximum and minimum exponents in the single and double IEEE precisions.

Parameter	Single	Double
Exponent width in bits	8	11
Exponent bias	+127	+1023
E_{max}	+127	+1023
E_{min}	−126	−1022

Table 1.2: Encodings of the special values and their meanings.

Exponent bits	Fraction bits	Meaning
All ones	all zeros	$\pm\infty$ (depending on the sign bit)
All ones	non zero	NaN (Not a Number)
All zeros	all zeros	± 0 (depending on the sign bit)
All zeros	non zero	denormalized numbers

that might arise from invalid operations like $+\infty - \infty$.

If the exponent field is zero and the fraction field is not zero then it represents a denormalized number which is defined in the standard as:"A nonzero floating-point number whose exponent has a reserved value, usually the format's minimum, and whose explicit or implicit leading significand bit is zero." The denormalized numbers are used to provide for a property called gradual underflow. The basic idea behind gradual underflow is to preserve the veracity of the mathematical relation $x - y = 0 \;\Rightarrow\; x = y$. In a system where gradual underflow is not implemented, if the difference of two numbers is less than the minimum representable normalized number then the result might be flushed to zero and the previous relation is not preserved. With gradual underflow, the difference is represented as a denormalized number and is not equal to zero, hence the relation is preserved.

The standard also defines extended precisions corresponding to the single and double precisions. For these extended cases the significand's precision (t) must be higher than that of the corresponding basic format as follows:

$$\begin{aligned} t_{ext} &\geq t_{bas} + \lceil \log_2(E_{max_{bas}} - E_{min_{bas}}) \rceil \\ t_{ext} &\geq 1.2 t_{bas} \\ E_{max_{ext}} &\geq 8E_{max_{bas}} + 7 \\ E_{min_{ext}} &\leq 8E_{min_{bas}} \end{aligned}$$

For double extended, this results in $t_{ext} \geq 64$ and the exponent field width being greater than or equal to 15 bits. In those extended precisions, the standard allows the use of redundant representations.

To understand the reason for having single, double as well as extended precisions we should clarify the meaning of the word precision as it is used in this context. The precision of a number is

the mathematical value of the unit in the least place (or *ulp* as it is sometimes abbreviated). For the 32 bits single format the *ulp* of the significand is equal to $2^{-23} \approx 10^{-7}$. So, we only have about 7 significant decimal digits in such fractions which is obviously not enough for some calculations. Double precision gives $2^{-52} \approx 10^{-16}$, i. e. 16 decimal places after the fractional point. Higher precisions are achieved by the use of the extended precisions defined by the standard which are usually handled by firmware. The quest for more precision in some calculations (specially scientific computations on supercomputers like the Cray machines) led the committee that is currently revising the standard to include a quad precision in the revision that should be published in the near future.

The standard has another feature that is worth explaining: the rounding modes. Any arithmetic operation must be carried out to give its result as if it was infinitely precise. Rounding is then applied to fit this precise result into the precision (single, double or extended) required by the user. The result is rounded according to one of the following four possible modes:

Round to Nearest Even, RNE: round to the value nearest to the infinitely precise result. If the two nearest representable values are at equal distances choose the one with the least significant bit equal to zero (the even number if viewed as an integer). This is the default rounding mode.

Round to Zero, RZ: round to the value closest to and not greater in magnitude than the infinitely precise result.

Round to $+\infty$, RP: round to the value closest to and not less than the infinitely precise result.

Round to $-\infty$, RM: round to the value closest to and not greater than the infinitely precise result.

Some applications, such as graphics and digital signal processing, might not need all these rounding modes, precisions or even the gradual underflow property. If a designer is implementing a special purpose system for such an application there is no need to comply to all those details from the standard and he/she can do whatever suits the application. On the other hand, the majority of the general purpose processors implement the standard with all of its details. For uniformity in the design of the datapath of the processor, designers usually choose to implement in the hardware the double precision (or the double extended precision) and then narrow the result to a single precision if this is what the user requires.

Kahan provides details on the status of the standard, features and examples in his lecture notes [6] while a recent interview with him [7] details the history of the standard.[3]

[3] The standard is currently undergoing revision and designers who are interested to know about new features or help the revision committee should consult their website at:

`http://grouper.ieee.org/groups/754/index.html`

Table 1.3: Frequency of floating point instructions on the MIPS architecture for five programs of the benchmark SPECfp2000.

FP instruction	applu	art	equake	lucas	swim	average	% to FP instructions
Load	11.4	12.0	19.7	16.2	16.8	15.22	0.35
Store	4.2	4.5	2.7	18.2	5.0	6.92	0.16
add	2.3	4.5	9.8	8.2	9.0	6.76	0.16
subtract	2.9	0	1.3	7.6	4.7	3.30	0.08
multiply	8.6	4.1	12.9	9.4	6.9	8.38	0.19
divide	0.3	0.6	0.5	0	0.3	0.34	0.01
other	0.7	2.4	1.8	5.0	0.9	2.16	0.05

1.4 Floating point operations

The major floating point operations are: add, subtract[4], multiply, divide, square root as well as loading and storing floating point numbers. These operations constitute the required operations in any system supporting the IEEE standard. In other systems where the standard is not used, those remain the operations most frequently implemented. Other elementary functions like log, sin, tan, sinh, ... are not always present in hardware implementations and their speed of execution is considerably slower.

The occurrence frequency of an instruction is defined as the number of times this specific operation occurs divided by the total number of occurrences for all instructions constituting a benchmark. The floating point instructions reported in a recent study of the instruction mix [8] in five programs is reported in Table 1.3. In an older study [9] on the same architecture the reported numbers were significantly different. In the older study, the add and subtract instructions account for about 40% of the floating point instructions. The share of the multiply is 37%, that of the divide is 3% and that of the square root is 0.33%. The move instruction in that older study account for about 10%. Studies of instruction frequency vary due to different factors but the most important among them is the compiler used and the kind of optimizations it made. However, studies seem to agree that that the addition/subtraction unit is the most heavily used part since it is used in addition, subtraction and sometimes format conversion as well. The multiplication unit follows as a close second. Although the divide instruction is not very frequent, if the division unit is too slow it can cause a large performance degradation for the whole system.

Due to these facts, the research presented here focuses on enhancing the performance of the addition and multiplication while preserving the performance of the division.

[4]The add and the subtract are done by the same piece of hardware since the floating point numbers are signed and adding two numbers with opposite signs constitutes an effective subtraction.

1.5 Layout of the research

In this work, a new internal format is proposed in chapter 2 to represent the floating point numbers within the FPU and its associated registers. This format uses a redundant representation and is based on the definition of the double format of the IEEE standard for floating point arithmetic. On loading a number into the FPU, it is transformed to this format then all the operations occur using it as long as the operands reside in the register file. If a store operation is issued, the number is transformed to the basic single or double precision as required. Chapter 3 details a parametric time delay model developed to compare the proposed architectural designs with state of the art implementations.

Correctly rounding the numbers according to all the modes of the IEEE standard proved to be one of the challenges in this work. The other related challenge is performing the leading digit detection in case of a subtraction. Both challenges and their solutions are outlined in chapter 4 where the design of the adder is presented and compared to other state of the art designs. The design of the multiplier follows in chapter 5. Then, the division operation and the elementary functions are discussed in chapter 6. Finally, the issues relating to integrating the whole floating point unit with the rest of the surrounding hardware system are presented in chapter 7.

Since research is never ending, chapter 8 presents some open issues for future work.

Chapter 2

Redundant alternatives

2.1 Why an alternative format

The quest for higher performance in FPUs and the attempt to reach the theoretical limits on the speed of computations has been the leading cause for advancements in designs in the past few decades. Several designers proposed the use of different formats for representing the numbers within the FPU in order to achieve better speed [10, 11]. Some implementations of commercial products (for example the x86 family of processors and their clones [12]) use internally a representation that speeds the processing while complying externally to the IEEE standard. In other cases, companies needed to retain other formats used by their machines while introducing the IEEE standard. The resulting implementation uses an internal format different than that of the standard. This latter case is exemplified by the current S/390 architecture [13].

An example of integer addition in the decimal system can help us explain some of the concepts that we need here. Consider the addition: $6789 + 3214 = 10003$. The value of least significant digit (3) of the result depends only on the values of the least significant digit of each of the two inputs. Namely, the 9 and the 4. On the other hand, the value of the most significant digit (1) depends on all the digits of the two inputs. Intuitively, if we implement an adder in hardware the output that depends on the largest number of inputs takes the longest time to be computed. Obviously, the type of technology used affects this time. So, if a logic gate accommodating a large number of inputs exists we can use it and finish the computation faster. The maximum number of inputs that a gate takes is called the fan-in. If large fan-in gates do not exist we need to build a tree of smaller gates to achieve the same logic function of the larger gate and incur a longer time delay. Those two factors, the number of inputs affecting any output and the fan-in of the gates in the technology are important in driving the decisions regarding circuit optimization for increased performance.

Winograd [14, 15] and later Spira [16] worked to formulate the theoretical lower limit on the time delay required to compute arithmetic operations. That limit is set by the largest number of

inputs (n) affecting any of the outputs. The limit also depends on the fan-in (f) of the logic gates in the technology used. Then the limit states that the time taken to perform the operation is bound by $\tau \geq \lceil \log_f n \rceil$ where the time is given in units of gate delays. Spira proved this limit for any function depending on n inputs and then commented "...this bound is extremely poor for almost all functions. Amazingly enough, however, it is a tight bound for addition and multiplication time and a good bound for division time."

Once a specific technology for implementation is chosen, the maximum fan-in is set. The way to speed on the algorithmic level thus becomes to decrease the number of inputs on which the outputs depend and to simplify the logic functions used. A number system where any digit of the output depends only on a small number of the input digits is then ideal to achieve the lower limit of time delay. Two such possibilities are discussed here: the Residue Number System (RNS) and the Redundant Representations for Numbers.

2.1.1 Residue Number System

The RNS [17] is based on the old Chinese number system where numbers do not follow the weighted positional formula $X = \sum_{i=0} x_i \beta^i$ that we presented in the previous chapter. In the RNS a few relatively prime numbers (i.e. with no common factors) are chosen to form the bases or the weights for the positions. For example, 9, 8 and 7 can be the bases for an RNS and each number is then represented by its residues with respect to those bases. So, 145, 166 and 311 are represented as

$$\begin{aligned} 145 \Rightarrow \{145 \bmod 9, 145 \bmod 8, 145 \bmod 7\} &= \{1, 1, 5\} \\ 166 \Rightarrow \{166 \bmod 9, 166 \bmod 8, 166 \bmod 7\} &= \{4, 6, 5\} \\ 311 \Rightarrow \{311 \bmod 9, 311 \bmod 8, 311 \bmod 7\} &= \{5, 7, 3\} \end{aligned}$$

Notice that $(5+5) \bmod 7 = 3$, $(1+6) \bmod 8 = 7$ and $(1+4) \bmod 9 = 5$ which means that adding the representation of 145 to that of 166 using the corresponding bases as moduli gives the representation of their sum, 311. Using RNS, each output digit depends only on the corresponding input digits and there is no carry propagation. This property makes it ideal from the speed of operation point of view for addition. The same property exists for subtraction and multiplication. However, the division and square root operations, comparing the magnitude of two numbers and sign detection are difficult operations in such a system. These operations can be done in a restricted form such as the division by a constant [18]. In general however, there is a need to convert from RNS representations to a weighted positional representation to accomplish those operations efficiently on any operands. The two main problems of RNS are:

1. The system is not complete and does not allow for all of the needed primitive operations.

2. Conversion to and from this system is often needed and it is a complicated and time consuming operation.

Due to these inherent limitations, the RNS is not widely used in general arithmetic circuits. It does have, however, specific areas of applications like cryptography [19] and digital filters [20, 21].

2.1.2 Redundant Representations

Another method for decreasing the number of digits on which any digit in the output depends is the use of a redundant representation. Redundant representations can be used in any number system. For example, in the old Roman system if the improper form "IIII" representing 4 is allowed along side with the canonical form "IV" then we get a redundant representation. In the Indo-Arabic system allowing more than ten different numerals at any digit position will result in a redundant representation as already noted in chapter 1. Redundancy can be even applied to the residue number system [22] and in the logarithmic number system [23] (which is yet another number system with which we are not dealing in this work). Redundancy appears also in the implementation of high performance multipliers and dividers [24]. In a multiplication unit, the Booth recoding is a redundant representation of one of the operand. Keeping the sum of the partial products in a carry save form is another exploitation of redundancy. The division units implementing the SRT algorithm use redundant numbers.

To understand why redundancy is of importance consider a simple example based on the signed digit numbers first introduced by Avizienis [25] and later expanded by Parhami [26]. Ordinary Signed Digit (SD) numbers [27, 28] represent a number in radix $\beta > 2$ with digits $x_i \in \{-\alpha, \cdots, 0, \cdots, \alpha\}$ where $\frac{\beta}{2} < \alpha < \beta$. When summing two digits: $p_i = x_i + y_i$, if $\pm p_i \geq \alpha$, the digit is recoded as $w_i = p_i \mp \beta$ and a carry c_i of ± 1. The final sum is then given by $s_i = w_i + c_{i-1}$

The condition on α guarantees that $-\alpha + 1 \leq w_i \leq \alpha - 1$. So the intermediate w_i sum can absorb any carry.

In the following example, we assume that $\beta = 10$ and $\alpha = 9$. Any negative digit is denoted with an over-line in order to clarify the representation, i.e. $1\,\overline{8} = (+1) \times \beta^1 + (-8) \times \beta^0 = 0\,2$. The left hand side represents an addition operation done using signed digits while the right hand side represents the same operation with a non-redundant representation.

	2	1	$\overline{8}$				2	0	2
+	7	9	9			+	7	9	9
	9	10	1	$\mid p \mid \geq \alpha$?			9	9	11
1	1	0		c					
	$\overline{1}$	0	1	w					
1	0	0	1	s		1	0	0	1

All the digit positions in the signed digit side can be done in parallel and the result obtained quickly. On the other hand, in the non-redundant representation the worst case carry propagates from the least significant digit to the most significant digit. In the case of signed digits as described above, the output digit depends on the input digits at two positions: the corresponding position and the adjacent lower order. This fundamentally changes the time delay for the operation. Instead of depending on all the digits, the most significant digit of the output, as well as all its digits, depend on a much smaller number of input digits. In fact, Kornerup [29] proves that converting from any number system using a redundant or non redundant digit set to another with a non redundant digit set is an operation of $\mathcal{O}(\log n)$, i.e. it takes an amount of time proportional to the log of the number of digits in the number. He also proves that conversion to a redundant digit set can be done in a constant time regardless of the number of digits in the number. Blair [30] takes the special case of converting redundant binary to two's complement numbers and shows that it is equivalent to two's complement addition. The work of Kornerup, on the other hand, is independent of the actual encoding of the digits and of the circuit implementation. He demonstrates that "arithmetic algorithms and their properties can be analyzed at the digit level, the actual encoding of digit value is only of concern when designing the actual circuitry."

The research of Parhami on generalized signed digit number systems [26, 31] concentrates, as Kornerup remarks, on the digit level. The same is true for another attempt at presenting a unified framework for redundant number representations by Phatak and Koren [32] which they called the hybrid signed digit. This unified framework includes the two's complement representation and the signed digit representation as special cases. The hybrid signed digit system, in contrast to the work of Parhami, allows some digits to be signed while others are unsigned and allows for a non uniform distance between the signed digits.

Some researchers did propose actual designs. A number of those concentrated on only the multiplication. As noted earlier, Booth recoding and the use of carry save adders are both introductions of redundancy but the researchers went further than that. Ferguson and Ercegovac designed a multiplier accepting its operands in a redundant representation [33]. Makino *et al.* implemented a $54\text{bit} \times 54\text{bit}$ multiplier using a redundant binary architecture and compared it with the conventional designs [34]. Takagi *et al.* also designed a multiplier with a redundant binary addition tree [35]. Ercegovac and Lang pioneered the technique that they called "on the fly conversion" enabling a fast method to get the most significant part of the multiplication result without carry propagation [36]. Other multiplier implementations included the use of hybrid signed digit representations [37], recoded binary signed digit [38] and signed binary digits [39].

Other researchers went beyond multiplication. Practically, any work using the SRT algorithm [24, 40] for division uses redundancy. Nielsen *et al.* presented a pipelined adder that accepts one of its operands in a redundant representation [41]. Edamatsu *et al.* implemented a multiplier and a divider using a redundant binary representation [42]. Piuri and Stefanelli proposed the use of redundant

binary representations for fault-tolerant arithmetic [43]. Ramamoorthy *et al.* studied the use of signed digits in digital signal processors architectures [44]. Morales proposed a convention for signed binary floating point [10]. Chen suggested a bit parallel arithmetic processor using a redundant binary representation [45]. Saed *et al.* went even further and proposed arithmetic with signed analog digits [46].

Some research was done specifically on the issue of converting between digit sets. Because of its wide use in multiplier circuits, Booth recoding is an important form of conversion into a redundant representation. Vassiliadis *et al.* studied and proved the correctness of the multiplication using the different schemes for Booth recoding [47]. Lyu and Matula looked at the Booth recoding of a redundant binary number [48] while Sam and Gupta studied the issue of multi bit recoding of a two's complement number in general and applied it to multipliers [49].

Yen *et al.* studied efficient conversion from redundant representations to binary numbers [50] on the logic level while Wey *et al.* implemented a self-timed circuit for the same problem [51].

Daumas and Matula studied a special form of recoding that allows a partial compression of the redundancy available in the original digit set [52, 53].

Going back to the problems that we found with the RNS, we see that redundant representations solve the problem of providing a complete system. The needed operations for a floating point unit (addition, multiplication and division) are possible in this case. In fact, various implementations have been proposed for specific units. Conversion time delay however remains an issue. Fundamentally, it takes an $\mathcal{O}(\log n)$ time delay and cannot be done faster [29]. The question then becomes:"Is it possible to design a system where the number of conversions is minimized or where the conversion is done in parallel with another needed operation?"

2.2 Overview of the proposed system

What is proposed here is a number format very similar to that of the IEEE standard. We represent every 4 bits of the significand redundantly as a signed digit number by using 5 bits in two's complement form. The fifth bit (extra bit) thus has the same mathematical value as the least significant bit (LSB) of the next higher group but opposite sign. This is shown for the string of bits $a4, a3, a2, a1, a0$ in Fig. 2.1. However, it is saved in the register file of the FPU to the right of that LSB and to the left of $a3$. The bits for G, R and S are those for the guard, round and sticky digits respectively since the numbers are saved to the register file unrounded. The reason for this is detailed later. The significand is always positive, as in the IEEE format, so there is no need for an extra bit in the most significant digit. It is also always normalized. Denormalized IEEE numbers are dealt with upon loading into the register file to normalize them. The definitions for infinities, NANs, and zeros are similar to the IEEE format. It is assumed that the base for the exponent is 16 and not 2 as the normal IEEE format. So, the number is given by $(-1)^{sign} \times 16^{exp-bias} \times first\,digit.remaining\,digits$.

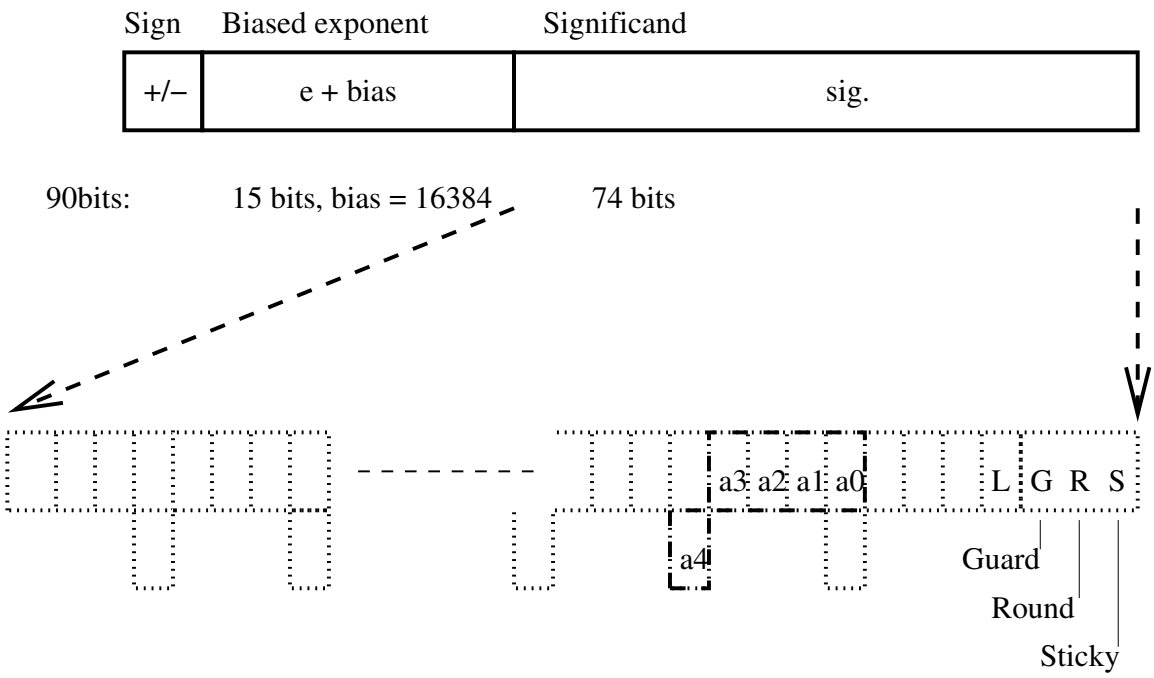


Figure 2.1: The proposed SD format for floating point numbers.

The basic idea is to use a redundant representation with signed digits instead of the standard format. The use of a hexadecimal-based exponent is not new, it was used in the IBM mainframes and survives in the current S/390 architecture [13] which supports both the hexadecimal and the IEEE standard formats.

2.2.1 Reasons behind the choice of field widths and exponent base

To explain the choices, a binary base for the exponent can be assumed first instead of the hex-base with an exponent field width of 17 bits and bias of 65535. The width of 17 is chosen so that any denormalized number of the IEEE standard can be put in a normalized form in the internal format. Since the IEEE double precision format has 11 bits in its exponent field and the least denormalized number might need to be shifted left by 52 bits, then an additional 6 bits ($2^5 < 52 < 2^6$) are necessary to accommodate it. In this case, when the numbers are loaded into registers, the IEEE format can be converted to the proposed format just by extending it with zeros as the additional bits in the significand to preserve the value of the number. The additional bits here mean the extra bit of each group of 4 bits to form an SD number as well as the trailing positions for single precision and the G and S bits. Then, the exponent bias needs to be adjusted accordingly. So, if the original number is in single precision, there is a need to subtract 127 and to add 65535. Such a binary-base

format would be a straight forward extension of the IEEE format. However, a higher base provides several positive results. For example, a hexadecimal-base is better than the binary-base because of the shifting needed for aligning the operands and for normalization in case of addition. If the exponent base is binary a shifter capable of shifting to any bit position is needed. On the other hand, if the exponent base is hexadecimal, only shifts to digit boundaries (4 bits boundaries) are needed. Obviously, with a hexadecimal-base, only 15 bits are needed to represent the same exponent range as the 17 bits needed for a binary-base. A higher base makes the shifting easier but it produces a less redundant number.

The choice of a hexadecimal-base is due to the different factors that affect the choice of the radix for the digits forming the numbers in the current system as well as the possible digit values.

- The carry-free addition and negation of SD numbers are simpler when the set of digits chosen is symmetric around zero such as $\{-\alpha, \cdots, -1, 0, 1 \cdots, \alpha\}$.
- Carry-free addition is possible if the number of different digit values ($2\alpha+1$ in the above case) is greater than the radix β by at least 2 when $\alpha > 1$ and $\beta > 2$ [26].
- If $\alpha < \beta$ then the sign of the SD number is the sign of the most significant digit and the number zero has only one representation which is all zero digits [31].
- When the design of the adders needed is considered, the use of $\alpha = \beta - 1$ becomes the easiest choice for implementation.
- If the value of the radix β is large then the carry delay within the digit becomes larger.

For all these reasons, the radix is chosen to be $\beta = 16$ (groups of 4 bits) and the digits are to be in $\{-15, -14, \cdots, 14, 15\}$. It is important to note that the radix of the digits in the significand does not necessarily have to be equal to the base of the exponent. For example an exponent base of 16 can be used with radix of 4 (2 bits) for the significand digits.

As for the size of the significand part, the literature about the precision of various floating-point number systems [54, 55, 56, 57, 58] defines two kinds of representational errors: the maximum relative representation error (MRRE) and the average relative representation error (ARRE). The terminology and notation for those errors in the different papers of the literature are not consistent. Hence, we use a simple notation where t is the significand bit width in a system with exponent base β and derive the equations giving those two quantities for any real number x. Then, we proceed to use them in our analysis of the binary and hex-based systems.

Let $x = f_x \times \beta^{exp}$ be an exact representation of x assuming that f_x has as many digits as needed (even an infinite number of digits if needed) but that f_x is normalized. This means that $1/\beta \leq f_x < 1$. Let the computer representation of x be $f_R \times \beta^{exp}$. Then the error of the representation is

$error(x) =\mid f_x\beta^{exp} - f_R\beta^{exp} \mid$. The MRRE is defined as the maximum error relative to x, i.e.

$$MRRE = max(\frac{\mid f_x\beta^{exp} - f_R\beta^{exp} \mid}{f_x\beta^{exp}})$$

If the exact number is rounded to the nearest representation then the maximum $error(x)$ is equal to half the unit in the last position (ulp) or $max(error(x)) = 1/2 \times 2^{-t} \times \beta^{exp}$. Thus,

$$MRRE = max(\frac{1/2 \times 2^{-t}}{f_x})$$

The denominator should be set to its least possible value which occurs at $f_x = 1/\beta$. Hence the MRRE is given by

$$MRRE = \frac{1/2 \times 2^{-t}}{1/\beta} = 2^{-t-1}\beta$$

In the derivation of the formula for ARRE, we use half of the maximum error since it is assumed that the error is uniform in the range $[-\frac{1}{2}2^{-t}\beta^{exp}, \frac{1}{2}2^{-t}\beta^{exp})$ for any specific $f_x\beta^{exp}$. As for the distribution probability of the numbers $f_x\beta^{exp}$ in the system, it is assumed to be logarithmic and given by $p(f_x) = \frac{1}{f_x \ln \beta}$. This assumption is based on the work of McKeeman [58] who suggested that "during the floating point calculations, the distribution of values tends to cluster towards the lower end of the normalization range where the relative representation error tends to be the largest." To get the ARRE we perform the integration

$$\begin{aligned} ARRE &= \int_{\frac{1}{\beta}}^{1} \frac{1/2 \times (1/2 \times 2^{-t})}{f_x} \frac{1}{f_x \ln \beta}\, df_x \\ &= \frac{2^{-t}}{4 \ln \beta} \int_{\frac{1}{\beta}}^{1} \frac{df_x}{f_x^2} \\ &= \frac{2^{-t}}{4 \ln \beta} \left[\frac{-1}{f_x}\right]_{\frac{1}{\beta}}^{1} \\ &= \frac{\beta - 1}{4 \ln \beta} 2^{-t} \end{aligned}$$

An analysis of both the MRRE and ARRE of the binary ($\beta = 2$) and hex-based ($\beta = 16$) systems reveals that more bits are needed in the case of hexadecimal digits in order to have the same or better relative errors. If $\beta = 2^k$ and the width is t_k the formulas for MRRE and ARRE can be written as:

$$\begin{aligned} MRRE(t_k, 2^k) &= 2^{-t_k-1}2^k \\ ARRE(t_k, 2^k) &= \frac{2^k - 1}{4k \ln 2} 2^{-t_k} \end{aligned}$$

To have the same or better error for a base $\beta = 2^k$ in comparison to the binary-base (2^1), the gaps

between two successive floating-point numbers in the larger base must be less than or equal to the gaps in the binary-base. So, if the exponent in base $\beta = 2^k$ is e_k then for base 2^1, $gap_1 = (2^1)^{e_1} 2^{-t_1}$. For base 2^k, $gap_k = (2^k)^{e_k} 2^{-t_k}$. It should be noted that $e_1 = e_k \times k + q$ where $\mid q \mid < k$. In fact with this definition, q is always a negative integer as illustrated by the following simplified example

		exp.	*mantissa*
$\beta = 16$		101	$0010xxxxxx$
$\beta = 2$	*before normalization*	10100	$0010xxxxxx$
$\beta = 2$	*after normalization*	10010	$10xxxxxx$

So, The potential left shift for normalization of up to $k-1$ positions makes q negative and it falls in the range $-(k-1) \leq q \leq 0$. Specifically, in the case of $k = 4$, $q \in \{-3, -2, -1, 0\}$. With that in mind, to have the same or better representation for the case of $\beta = 2^k$ the following must hold:

$$\begin{aligned} gap_k &\leq gap_1 \\ (2^k)^{e_k} 2^{-t_k} &\leq (2)^{e_1} 2^{-t_1} \\ ke_k - t_k &\leq e_1 - t_1 \\ ke_k - (ke_k + q) &\leq t_k - t_1 \\ -q &\leq t_k - t_1 \end{aligned}$$

In order to have the last inequality true for all the values of q then

$$t_k - t_1 \geq k - 1$$

If t_k is chosen to be $t_1 + k - 1$ and then substituted in the equation for MRRE, the maximum relative representation error becomes

$$\begin{aligned} MRRE(t_k, 2^k) &= 2^{-t_k - 1} 2^k \\ &= 2^{-(t_k - (k-1))} \\ &= 2^{-t_1} \end{aligned}$$

which is intuitive. Equal gaps in both systems means that the same set of numbers out of the real numbers range is being represented in both systems and hence the maximum representation error must be equal.

The average relative representation errors on the other hand are not equal because of the different distribution probability of the numbers. The ratio of $ARRE(t_k, 2^k)$ to $ARRE(t_1, 2^1)$ is

$$\frac{ARRE(t_k, 2^k)}{ARRE(t_1, 2^1)} = \frac{2^k - 1}{k 2^{k-1}}$$

$$= \frac{2}{k}(1 - \frac{1}{2^k})$$

So, for all $k > 1$, $ARRE(t_k, 2^k) < ARRE(t_1, 2^1)$.

For the case of $k = 4$ or $\beta = 16$ this analysis reduces to $t_{\beta=16} \geq t_{\beta=2} + 3$. Since for the IEEE standard, the double precision has $t_{\beta=2} = 53$ then $t_{\beta=16}$ is chosen to be 56 bits or 14 hexadecimal digits. Each of these digits requires an additional bit for its sign except for the most significant digit which is always positive. The guard, round and sticky digits are represented by 5 extra bits. So, at the end, the significand needs $56 + 13 + 5 = 74$ bits which is what is shown in Fig. 2.1.

Another extended precision can be defined similar to the extended precision of the standard. For the extended double precision of the standard, a minimum of 15 bits is required for the exponent and a minimum of 64 bits for the significand. Using a similar analysis to what was presented above, the width of the hex-based exponent field is at least 19 bits and $t_{\beta=16} \geq 67$. Since the digits are four bits, the minimum $t_{\beta=16}$ is really 68 bits.

2.2.2 Conversion issues

On loading a number from memory the special cases of infinity, zero and NaN must be detected and encoded properly. For the general case, to convert from the IEEE format to the presented format both the significand and the exponent parts are modified. If the exponent of the IEEE format has N bits, its bias is given by $(2^{N-1} - 1)$. In the proposed format the bias is 2^{14}. If R is the remainder of the division of the biased exponent exp of the IEEE format by 4 then

$$\begin{aligned} 2^{(exp-(2^{N-1}-1))} \times 1.xxx\cdots &= 2^{(exp-2^{N-1})} \times 1x.xx\cdots \\ &= 16^{(\lfloor \frac{exp}{4} \rfloor - 2^{N-3} + 2^{14} - 2^{14})} 2^R \times 1x.xx\cdots \end{aligned}$$

Hence the new biased exponent for the hex-base is $\lfloor \frac{exp}{4} \rfloor - 2^{N-3} + 2^{14}$. The value of N is 8 for single precision and for double precision $N = 11$.

This amounts to shifting the exponent exp by two bits to the right then adding a one in the new bit position 14 (the new LSB is bit position 0) and subtracting another in bit position $N - 3$. Or equivalently, adding a string of all 1's from bit position $N - 3$ to bit position 13. There is no need to really perform an addition as shown in this example for the case of single precision numbers:

$$\begin{aligned} exxxxx & & exxxxx \\ -100000 &= +011111111100000 \\ +100000000000000 & \end{aligned}$$

Which gives: $e\overline{eeeeeeeee}xxxxx$. The hex exponent has the same bits as the shifted binary in positions $4-0$, then positions $13-5$ are the complement of the original exponent's MSB and finally

bit 14 is equal to the original exponent's bit MSB. A similar conversion occurs for the case of double precision. This function can be called *bthe* as a short notation for Binary To Hex Exponent function. As for the significand, if the original number is normalized in the IEEE format then four cases are possible:

$$\begin{array}{lcl} R=0 & 001x.xxxx\cdots & \times 16^{(bthe(exp)-2^{14})} \\ R=1 & 01xx.xxx\cdots & \times 16^{(bthe(exp)-2^{14})} \\ R=2 & 1xxx.xx\cdots & \times 16^{(bthe(exp)-2^{14})} \\ R=3 & 0001.xxxxx\cdots & \times 16^{(bthe(exp)-2^{14}+1)} \end{array}$$

For the case of $R = 3$ there is an added 1 in the exponent. If the original number is denormalized then the leading non-zero digit must be found and the number left shifted to normalize it. The new exponent would thus be equal to $bthe(0) - shift\ amount$. So, in general the exponent is given by:

$$exp_{16} = \begin{cases} bthe(exp_2) & \text{normalized and } R = 0, 1, 2 \\ bthe(exp_2) + 1 & \text{normalized and } R = 3 \\ bthe(exp_2) - shift & \text{denormalized} \\ \text{all ones} & exp_2 \text{ is all ones} \\ \text{all zeros} & \text{number is zero} \end{cases}$$

So, conversion from the normal binary formats to the proposed SD format is quite simple and at maximum would involve a short addition in the exponent to normalize denormalized numbers.

No conversion back is required as long as the number remains in the register file of the FPU. On the occurrence of a store to memory operation, the number is converted by subtracting all the extra bits of each group of 4 bits from the number. This is because the digits are represented in two's complement form and the two's complement has the property described in the following Lemma.

Lemma 2.2.1 *If a binary number x is represented in two's complement form by the bit string $x_n x_{n-1} \cdots x_1 x_0$, then $x = (-1)x_n 2^n + \sum_{i=0}^{n-1} x_i 2^i$.*

Proof:

- If x is positive, then $x_n = 0$ and the statement is true by the definition of binary number representation.
- If x is negative, then $x_n = 1$ and the absolute value of x is equal to the one's complement of the $x_n x_{n-1} \cdots x_0$ bits plus one. This means:

$$x = -(\sum_{i=0}^{n}(1 - x_i)2^i + 1)$$

$$
\begin{aligned}
&= -(1-x_n)2^n - \sum_{i=0}^{n-1}(1-x_i)2^i - 1 \\
&= 0 - \sum_{i=0}^{n-1} 2^i + \sum_{i=0}^{n-1} x_i 2^i - 1 \\
&= -2^n + 1 + \sum_{i=0}^{n-1} x_i 2^i - 1 \\
&= (-1)x_n 2^n + \sum_{i=0}^{n-1} x_i 2^i
\end{aligned}
$$

The fact that $x_n = 1$ in this case was used in the last line above.

This concludes the proof since in both cases, the statement is true.$\diamond$

So, the MSB of a 2's complement number is treated as if having a negative value. Hence, all the extra bits are to be subtracted from the number to convert it back to normal binary. This subtraction does use a normal carry propagation adder and is the place where the conversion price of a time delay of $\mathcal{O}(\log n)$ is paid. The following section, however, discusses how this delay can be hidden. The exponent also needs a conversion similar but opposite to the one needed while loading the number into the floating point unit.

2.3 Potential gains from the proposal

Now is the time to answer the question asked before the overview of the system: "Is it possible to design a system where the number of conversions is minimized or where the conversion is done in parallel with another needed operation?"

Two points are important to help us answer:

- The conversion back to the normal IEEE format takes place only when a register is to be stored in the main memory or to a write buffer outside of the floating point unit.
- With the current trends in processor designs, compilers optimize the code in order to have the operands in the registers most of the time.

These points then indicate that the conversion happens at the frequency of the store operations. This frequency can still be as high as 16% of the total number of floating point operations as indicated in chapter 1. The first part of the question (minimizing conversions) is thus answered but not in a very satisfying manner. The second part thus remains: can we hide the conversion delay with the delay of the store operation? Looking at the steps performed to achieve the store operation in a pipelined processor we see that it consists of:

1. Fetch the operation
2. Decode it to realize that it is a store.
3. Load the operands from the register file. Those operands are the content of the register to store in memory and possibly the content of another register that is used as an index for address calculation.
4. Calculate the address. This is the execution stage in other regular operations. If an immediate addressing mode is used (where the exact address is embedded in the encoding of the instruction), the processor is idle in this step.
5. Write to the memory. This could be through a cache mechanism, so we will assume here that it takes just one step and does not stall the processor.

What we notice here is that in step 4 the store operation is not using the floating point unit at all. The address calculation is done using an integer adder and nothing operates on the floating point number that is to be stored in the memory. It is during this time that we propose for the conversion to occur. That entails adding a piece of hardware to operate on the floating point number in parallel with the address calculation in order to remove the redundancy, round the number and put it in the appropriate IEEE format.

If the processor issues its instructions in order, this proposed scheme completely hides the conversion delay within the necessary delay of calculating the address in the store instruction. On the other hand, if the processor issues its instructions out of order there is a chance that the address calculation step of the store is executed before the floating point result that is to be stored in the memory becomes ready. In such a case, the conversion delay is not hidden within the store and it actually adds some time delay. The probability of this event obviously depends on the frequency of the store, the distance in the program code between the store and the instruction producing the output result, the rearrangements done by the processor and possibly other factors as well. The added time delay might not even affect the following instructions if they are independent of the store (i.e. they are not a load instruction trying to get the same result back from the memory) and they are allowed to commit their results and end their execution before the store ends. Another method of hiding the conversion delay in an out of order processor is to do it in parallel with the translation from a virtual address to a real address in the cache and memory control units. So, in out of order machines, the conversion delay might not always be possible to hide within the store. However, because it is an out of order machine, other opportunities for hiding that delay arises and can be considered in each specific architecture.

In conclusion, redundant representations allow us to easily perform the needed floating point operations. The conversion back to the IEEE standard format is the point where an $\mathcal{O}(\log n)$ time delay occurs but usually it can be hidden within the store operation.

Chapter 3

Modeling reality

3.1 What is "reality", why do we model it and why do we use parameters?

Today, digital circuits are implemented mostly in Silicon using CMOS technologies. The minimum feature size determines the minimum size of a transistor. This and the number of metal layers available for wires as well as their electric properties are probably the most important aspects that differentiate between CMOS technologies. They dictate to a large extent the speed of operation and the area of a given functional unit. As the feature size decreases the logic gates get faster. If there is not enough layers of wiring then the area of a basic cell is increased to allow for more wires to pass adjacent to each other instead of on top of each other. If there is a need for long wires to communicate across the chip then propagation delay is a factor affecting the speed of operation. The percentage of this wire propagation delay to the time it takes the logic gates to process the signal increases as the technology is scaled to smaller feature sizes [59].

The preceding paragraph constitutes reality in CMOS circuits. Other technologies are not discussed in this work. We restrict ourselves to static CMOS gates and CMOS pass gates. Switch transistors can be used inside multiplexers and shifters but we do not consider the general use of switch logic. Dynamic circuits, Domino logic and other circuit styles are not generally used in commercially available floating point units. Our goal is to compare our proposed redundant digit system to the state of the art systems and hence we restrict ourselves to what those systems use to have a fair comparison. If in the future faster circuit styles are implemented in floating point units, the increase in speed should be uniform on all the designs unless one of them relies on specific features of the older circuit style.

One might ask then why is modeling needed? Is it not possible to fabricate the new design and test it to see how it compares to other designs? Such a fabricated circuit would be really the

ultimate test to check the veracity of any claims made about a design. However, this is a very costly thing to do every time a designer considers a new idea. Floating point units are used in general purpose processors, dedicated hardware for graphics and in digital signal processors. For a certain application and design specifications (speed, area and power consumption), it may be necessary to compare several architectures. A simple model is needed to help in targeting a design for use at a different operand width (for example single and double precision), with a different hardware library or with a different radix and number system. In the following section we develop such a model that we use later to compare the proposed system to other designs.

The parameters of our model are the operand width, the logic gate fan-in and the radix of the redundancy for designs using redundant representations. In special hardware for digital signal processing or graphics applications, the operand format does not have to conform to a standard and is usually dependent on the application and on the other design restrictions. The significand may be 16 bits or less. Most general purpose processors conform to the IEEE standard which specifies certain formats with specific operand widths. The hardware designer who already designed a floating point unit for one format of the standard or for a specific application in graphics or signal processing may find that the design is not optimal for another format. The proposed parametric model can be used to identify the best architectures based on the given constraints. The range of desirable operand widths, (n), is quite large spanning from about $n = 8$ bits to about $n = 256$ bits depending on the application. Architectures that are beneficial for small operand widths may not be suitable for large operand widths.

The logic gate fan-in (f) is the second parameter to consider. A large number of designs use standard libraries with pre-designed components. The maximum fan-in of the standard gates is limited. For CMOS technologies the fan-in limitation is due to the number of series transistors that may be stacked in one pull-up or pull-down chain. Typical values for the maximum fan-in are between about $f = 2$ and $f = 4$ inputs per gate. In dynamic circuits the fan-in might be up to $f = 6$ inputs per gate.

The third parameter used in the model is the radix (2^r) of the digits. Signed digits or other forms of redundancy may be used to improve the unit's performance as discussed in chapter 2. In such designs, the operand width is divided into digits with some redundancy. The radix of the digits can be as small as 2 (binary) or as large as the operand width (non-redundant). Practical values for the radix range from radix 2 to about 256 or digit width between 1 and 8 bits.

In this chapter, we develop the model and then use it in chapters 4 and 5 to compare the proposed adder and multiplier to other designs.

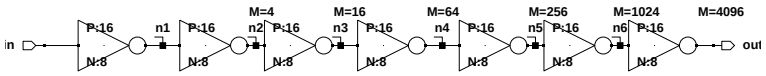


Figure 3.1: In CMOS $0.6\mu m$ technology, the pull down time is $0.40ns$ and the pull up time is $0.44ns$. Hence, the *FO4* is taken as $0.42ns$. In $0.3\mu m$ technology the numbers are $0.19ns$, $0.18ns$ and $0.185ns$ respectively.

3.2 The time delay model and its validation

The model proposed here gives an estimate of the number of equivalent elementary delay units in the critical path of the floating point hardware. The floating point unit delay is presented in "fanout of 4" delays, or the delay of an inverter driving a load that is four times its own size. This is commonly abbreviated as *FO4* for the "fanout of 4" inverter.

The simulation tool *irsim* (a switch level simulator for transistors) is used to simulate a number of circuits in order to validate the model. All the circuits are designed in a CMOS $0.6\mu m$ technology. The first such circuit is a chain of inverters properly scaled so that each one is four times the size of the preceding one as shown in Fig. 3.1. The chain is used to estimate the time delay of an *FO4* inverter. The pull down time of the inverter is $0.40ns$ while the pull up time is $0.44ns$. Hence, the average delay unit is estimated to be $0.42ns$.

In this model, for any integer adder the following simplified formula giving the gate delay of conditional sum addition [17] is used:

$$\tau = 5 + 2 \times \lceil \log_{f-1}(\lceil \frac{n}{f} \rceil - 1) \rceil$$

In the formula, n is the number of bits in the adder and f is the fan-in or the maximum number of inputs for a gate in the design. A $[4:2]$ compressor that adds four input bits plus a carry to produce two output bits and a carry is assumed to take 3 *FO4* delays while a $(3,2)$ counter that adds three input bits giving two output bits takes 2 *FO4* delays [60].

A single m-to-1 multiplexer is considered to take only one *FO4* delay from its inputs to the output assuming it is realized using CMOS pass gates. This assumption for the multiplexer is valid up to a loading limit. Small m is the usual case in VLSI design since multiplexers rarely exceed say a 5-to-1 multiplexer. Using *irsim* the 2-to-1, 3-to-1 and 4-to-1 multiplexers are simulated. They all exhibit a time delay from the inputs to the output within the range of one *FO4* delay (i.e. less than $0.42ns$).

When the input lines are held constant and the select lines change, the delay from the select lines to the output is between one and two *FO4* delays. Hence, for a single multiplexer the delay from the select lines to the output is bounded by 2 *FO4* delays. A series of m to 1 multiplexers connected to form a larger n-bit multiplexer heavily loads its select lines. Hence there is even a larger delay

from the select lines to the output in this case. To keep up a balanced design with a fanout of four rule, each four multiplexers should have a buffer and form a group together. Four such groups need yet another buffer and form a super group and so on. The delay of the selection then is assumed to be $\lceil \log_4(n) \rceil + 1$. This last formula is applicable even in the case of a single multiplexer since it yields 2 as given above.

The carry free signed digit adder used in the proposed designs is a number of parallel adders each taking digits composed of $r + 1$ bits ($radix = 2^r$) and adds them producing their sum, sum plus one and sum minus one. Then a choice is made between those three outcomes. Because of the more complicated carries in this scheme it is assumed that they take an additional *FO4* delay. The choice of which outcome of the adder to produce is basically a multiplexer that has a delay from its select line to its output and then there is an additional *FO4* delay for the last correction. So, the total delay of the signed digit adder in *FO4* delays is

$$\begin{aligned} \tau &= (5 + 2 \times \lceil \log_{f-1}(\lceil \frac{r+1}{f} \rceil - 1) \rceil) + 1 + (\lceil \log_4(r+1) \rceil + 1) + 1 \\ &= 8 + 2 \times \lceil \log_{f-1}(\lceil \frac{r+1}{f} \rceil - 1) \rceil + \lceil \log_4(r+1) \rceil \end{aligned}$$

This is a conservative estimate for the signed digit adder. Running over 100000 random test vectors, it is found that such an adder with $r = 4, f = 3$ and composed of three digits has a delay of $4.0ns$ using the $0.6\mu m$ technology. This delay is less than the 10 *FO4* delays predicted by the above formula.

Shifters can either be done by a successive use of multiplexers or as a barrel shifter realized in CMOS pass transistors. In either case, the delay of an n-way shifter from its inputs to its outputs takes $\lceil \log_2(n) \rceil$ *FO4* delays. The select lines are heavily loaded as in the case of multiplexers. However, if the same idea of grouping four basic cells is used then the delay from the select lines is the same as for the multiplexers. This is smaller than the delay from the inputs to the outputs in the shifter. Hence the input to output delay dominates and is the only one used. A 16-way shifter is designed using NMOS transistors and simulated with *irsim*. The model predicts that its delay must be less than $\lceil \log_2(16) \rceil = 4$ *FO4* delays. Using a set of random inputs to stimulate the simulation, the time delay from the inputs to the outputs is found to be less than $1.2ns$. This delay is equivalent to 3 *FO4* so the model is also conservative in this case.

For other pieces of combinational logic where a specific design is reported in the published papers, the delay can be estimated. If the design is not known, and the logic has n inputs then its time delay is assumed to be $\lceil \log_f(n) \rceil$ *FO4* delays. The different parts of the model are summarized in Table 3.1.

Using units of *FO4* delays makes the model independent of the technology scaling to a large degree since this elementary gate scales almost linearly with the technology [59]. Such units also make the model take into effect the time delay associated with the small local wires inside the *FO4*

Table 3.1: Time delay of various components in terms of number of *FO4* delays. f is the maximum fan-in of a gate and n is the number of inputs.

Part	Delay
Adder	$5 + 2 \times \lceil \log_{f-1}(\lceil \frac{n}{f} \rceil - 1) \rceil$
$[4:2]$ compressors	3
$(3,2)$ counters	2
Multiplexer, input to output	1
Multiplexer, select to output	$\lceil \log_4(n) \rceil + 1$
Signed digit adder	$8 + 2 \times \lceil \log_{f-1}(\lceil \frac{r+1}{f} \rceil - 1) \rceil + \lceil \log_4(r+1) \rceil$
Shifter	$\lceil \log_2(n) \rceil$
Other (no design details)	$\lceil \log_f(n) \rceil$

inverter as well as those connecting it to neighboring gates. However, the model does not include any assumptions about long wires across the chip and the time delay associated with them. Hence, obviously, it does not give an accurate estimate of the absolute delay of a logic unit. However, the model can be used to compare different architectures to estimate their relative speeds. The reason that the model does not differentiate between the delay time of the different types of gates is that the designers usually change the sizes of the transistors in order to equalize the time taken by all gates on the critical path.

3.3 Levels of evaluation

Modeling at the logic gate level as described above does not capture all the details of real CMOS circuits. Design evaluation is an iterative process with several levels of complexity. At each level different ideas are compared and the most promising are tried at the following level of complexity. The levels that are proposed here are:

1. Modeling at the logic level just as described above. This does not provide very accurate estimates and can be used for rough comparisons.

2. Implementing the design in transistors and simulating. This level forces the designer to think about sizing the transistors and to buffer any gates that are driving a large fan-out. This level gives a much more accurate estimation of the time delay but it still does not include the long wire delays.

3. For more accuracy, a layout of the full circuit can be done to extract the details about the wires. An extracted circuit (or at least its critical path) can then be simulated to give a more accurate time delay estimate. Area and power consumption estimation are also possible at this level.

4. The design is actually fabricated and the chip is tested. This is the ultimate test for a proposed design with a specific technology process and fabrication facilities.

5. To really show the merit of a proposed idea, the whole design can be simulated over a variety of scalable physical design rule sets and one or more are fabricated and tested.

The model proposed in this chapter is used primarily to evaluate the general trade-offs in the designs as illustrated in the following chapter. In chapters 4 and 5, the results of the first two levels of evaluation for the proposed adder and multiplier will be reported.

Chapter 4

Addition unit

4.1 Conventional adder designs and their time delays

In the current state-of-the-art high performance floating point adders, two-path algorithms [61] are used with integrated rounding [62] as shown in Fig. 4.1. These adders perform both addition and subtraction. An effective subtraction occurs when both operands are of the same sign and the required operation is a subtraction or when the operands differ in their signs and the operation is an addition. In the case of effective subtraction and an exponent difference of zero or one, a number of the leading bits of the result might become zero. In this case, a left shift of the result is needed for normalization. The two-path algorithm separates this specific case into a path with a specialized left shifter while other cases of operands pass through the regular path. The special path is called the cancellation path (where the leading digits are possibly canceled) or the close path (where the exponents are close to each other) while the regular path is called the far path. All floating point adders include circuits to either detect or predict the position of the leading non-zero digit after the subtraction is performed. The prediction circuits [63, 64] operate on the adder's operands in parallel with the significand addition. Prediction circuits are obviously more complicated than detection circuits but they improve the overall adder time delay. In the far path, the right shift is performed on the operand with the smaller exponent to align the two numbers and equalize their exponent. The bits that are shifted out are used to determine the rounding decision. If the final adder produces both the sum and the sum plus one it is called a compound adder. To speed up the rounding, a compound adder is used and one of its outputs is selected as the rounded result [65].

Fig. 4.1 shows yet another possible improvement using a variable latency adder [66, 9]. If the close path is chosen and the output does not need to be left shifted, that output is then ready after just the first cycle (cycles are marked by the dashed line in the figure). If it needs a left shift as indicated by the Leading One Predictor, LOP, and the priority encoder, PENC, it is available after the second cycle otherwise it takes three cycles to finish. A collision detection circuit can be used to

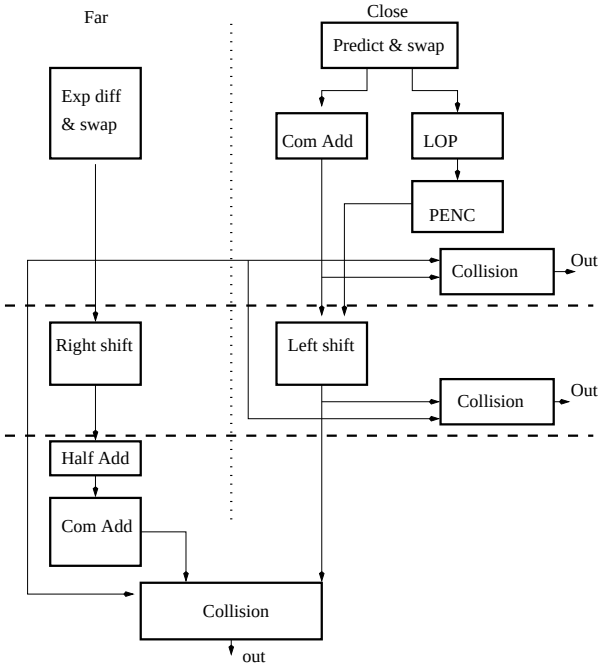


Figure 4.1: The conventional two-path adder.

prevent two outputs from getting out of the pipelined adder on the data bus at the same time. Some recent attempts for improving the adder and simplifying its design include eliminating the need for rounding in the close path [67]. To explain this idea, let us assume that the close path is only chosen in the case of effective subtraction with all the additions regardless of the exponent difference going to the far path. In this case, being in the close path, one of the operands is shifted at most by one bit location. Hence, the result might extend at most by one bit on the least significant side. Having only subtractions means that there can never be an addition overflow and there is never a need to right shift the result for normalization. The MSB of the result is thus at most aligned with the MSB of the inputs. There is still a need for a left shifter to normalize the number in case of a cancellation of the most significant bits. If such a cancellation occurs and a left shift is needed, even by one bit location, then the result is guaranteed to be accurately represented within the precision of the format used and no rounding is necessary. Hence, if the conditions of using the close path become:

- exponent difference is equal to zero or one.
- only effective subtraction
- cancellation does occur

then this set of conditions guarantees that there is no need for rounding logic in the cancellation path and the adder used there could be simpler to design and possibly slightly faster. A recent

study [68] looked at various floating point adder designs from the literature and categorized the different optimization techniques used in them.

The simple model presented in the previous chapter allows us to have a sense of the complexity of some parts of the floating point addition algorithms. Shifting and adding are operations whose time delay is an $\mathcal{O}(\log n)$. In conventional designs, rounding adds a small value to the result and could cause a carry propagation making it an $\mathcal{O}(\log n)$ operation. It is usually combined with the addition step [62] so that both time delays overlap. Another $\mathcal{O}(\log n)$ delay corresponds to the leading one prediction [64]. Although a redundant format that makes the significand addition independent of n enhances the speed (slightly), improvements in the other delay factors such as postponed rounding and the other features in the proposed system are integral to the design and are as important as the redundant format. To quantify this argument, the proposed design for the adder is described in the following section, then the assumptions of the analytical model are used to estimate its time delay.

4.2 Proposed adder design

The adder design presented here follows a conventional adder approach and uses a two-path algorithm as shown in Fig. 4.2. The cancellation path is used only in the case of an effective subtraction with an exponent difference of zero or an effective subtraction with an exponent difference of one and a cancellation of some of the leading digits occurring in the result. In all other cases, the far path is used. As explained above, these conditions simplify the cancellation path.

The far path of the proposed adder differs from the far path of conventional adders in some unique aspects: first, the use of a hexadecimal base for the exponents; second, the location of the rounding logic in parallel with the exponent difference and third the use of signed digit numbers in the significand. The hexadecimal base of the exponents makes the right shift for alignment a shift to a 4-bit boundary only. So, instead of using an n-way shifter in the conventional adders an $\lceil n/4 \rceil$-way shifter is used here. Such a reduction in the complexity of the shifter reduces its time delay as discussed below. The parallel execution of the rounding logic with the exponent difference logic takes the rounding away from the critical path of the adder. Our design simultaneously rounds and negates the number to prepare the operand for the SD (signed digit) adder. The rounding block is described in section 4.2.2.

As presented in Fig. 4.2, the far path proceeds as follows:

1. The rounding blocks produce the positive and negative rounded number corresponding to each operand.

2. A signal indicating an effective subtraction selects the operand or its negative.

3. Another signal indicating the operand with the larger exponent permits the swapping of the operands.

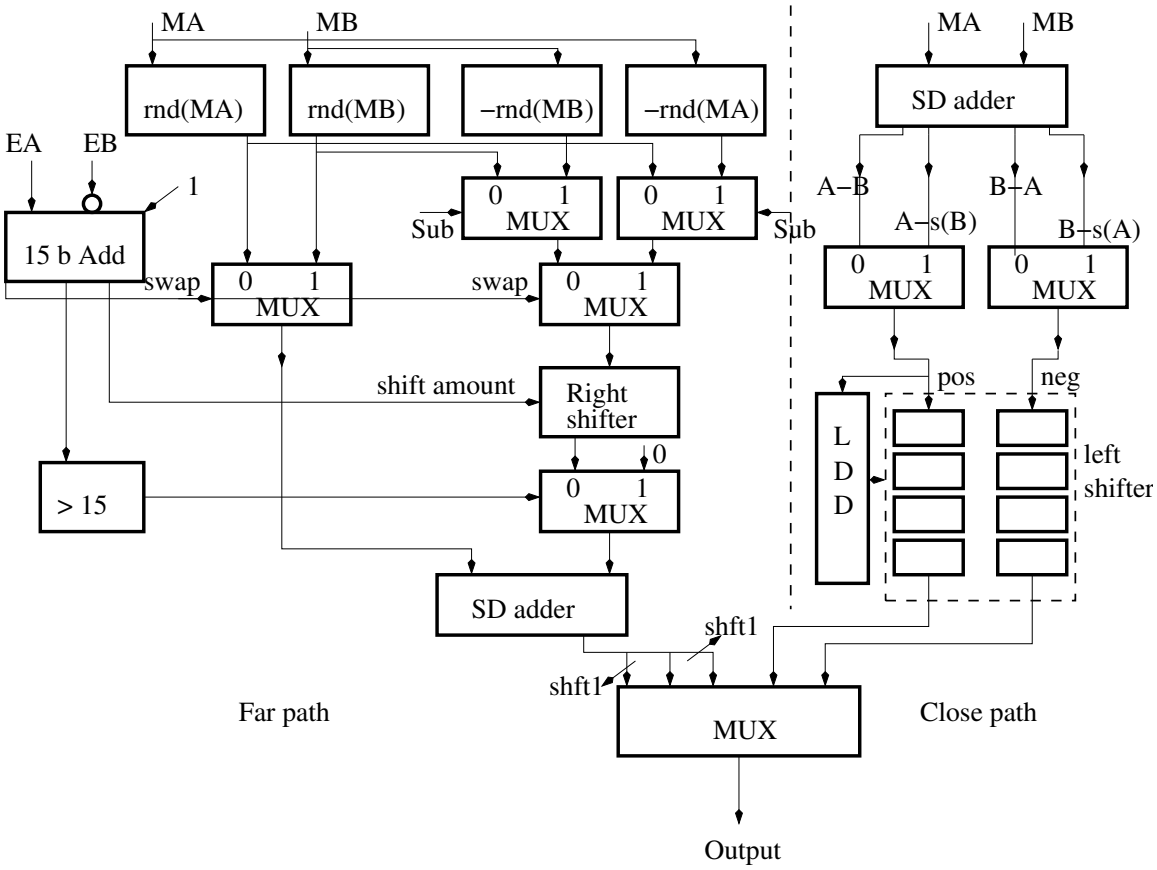


Figure 4.2: Block diagram of the two-path adder.

4. The operand with the smaller exponent is shifted to the right for alignment and the bits that are shifted out are used to calculate the guard, round and sticky bits. Only the least significant bits of the exponent difference are used to indicate the shift amount.

5. If the exponent difference is large enough to completely shift out the smaller operand a zero is forced as the second operand into the adder.

6. The SD adder is used to add the operands.

The result of the SD adder may need a normalization shift by one digit to the right for the case of effective addition and an overflow. The result may otherwise need a normalization shift by one digit to the left for the case of an effective subtraction and cancellation of the Most Significant Digit (MSD). This cancellation of only the MSD can occur even when the exponent difference is larger than one. The SD adder block produces the result and three signals indicating the need for no shift, a shift to the left or a shift to the right. The guard, round and sticky bits are calculated speculatively dependent on the shifting possibilities. The multiplexer unit responsible for choosing between the far and cancellation paths receives those different signals and speculative results and chooses the final result among them in case the far path is chosen.

In the cancellation path the exact exponent difference is not calculated but the least significant bit of each of the two exponents is examined. If the two exponent bits are found to be identical the difference of the exponents is speculated to be zero and the direct subtraction of the operands is performed. If, on the other hand, the two exponent bits are not identical the difference is assumed to be one and the subtracter produces a result equal to one operand minus the other operand shifted by one bit to the right. The direct subtraction in the case of zero exponent difference may lead to a negative result if the significand of the second operand is larger than the significand of the first operand. To remedy this negative result in conventional adders, the sign of the floating point result is flipped and the bits representing the result are negated. In our redundant digit design, the subtracter produces the result and its negative and then one of them is chosen at the end depending on the sign of the result.

So, for two operands labeled A and B, the calculation in the cancellation path proceeds as follows:

1. All the possible combinations are produced: $A - B$, $A - shift(B)$, $B - A$ and $B - shift(A)$.

2. Since a complete calculation of the exponent difference does not occur in the cancellation path, the rounding is done in conjunction with the significand subtraction. A round digit is computed for each operand and is used within a signed digit subtracter to perform the subtraction.

3. Depending on the exponent difference, either the direct subtraction or the one involving a shift is chosen.

4. A Leading Digit Detector (LDD) is used to calculate the shift amount needed to normalize the result.

5. This shift amount is applied to two shifters, one shifting the result and the other shifting the negative of the result.

6. The sign of the leading digit is checked to determine the correct sign for the result.

The result and its negation as well as a signal indicating the sign of the leading digit are forwarded to the multiplexer unit that selects between the cancellation path and the far path. This multiplexer unit then selects the appropriate output.

4.2.1 First challenge: The leading digit detection

As noted above, leading digit prediction circuits [63, 64] operate on the operands of the adder in parallel with the significand addition while detection circuits operate on the result. A recent study [69] compared a number of the designs proposed for both prediction and detection. This study shows that the predictors generate a string of bits having approximately the same number of leading zeros as the output of the adder. Then a detection scheme is used on that string to encode the location of the first non-zero element and control the normalization shifter.

In a number system that uses non-redundant digits, the first non-zero digit is the leading digit in a normalized number. However, if a redundant format is used a few patterns may change the location of the leading digit. This has been reported in the case of the prediction scheme proposed by Quach and Flynn [64] as well as the work of Bruguera and Lang [63]. In both schemes the original operands are not redundant while the prediction circuits are working on a redundant representation because the prediction is done before the result of the adder is available. If a detection scheme was used in those two cases, there is no need for complicated pattern matching. In the case of the proposed signed digit format, the operands are redundant and the result of the adder is also redundant. So, even in a detection scheme pattern matching is required.

A few possible patterns become the hard cases in detecting the first non-zero digit. In fact, the leading zeros may be expressed directly as zeros or indirectly as leading insignificant digits: a leading 1 followed by -15s or a leading -1 followed by 15s. The leading 1 (-1) can be converted to a zero and borrowed into the neighbor -15 (15) digit position as a 16 (-16). Since $16 - 15 = 1$ ($-16 + 15 = -1$), the zero propagation may continue into lower significance digits. The following example illustrates how leading non-zero digits may be actually representing leading insignificant digits. Assuming $\mid l \mid < 15$,

$$\begin{array}{rrrcrcccccccccccc} 1 & -15 & -15 & \cdots & -15 & l & m & \cdots & = & 0 & 0 & 0 & \cdots & 1 & l & m & \cdots \\ -1 & 15 & 15 & \cdots & 15 & l & m & \cdots & = & 0 & 0 & 0 & \cdots & -1 & l & m & \cdots \end{array}$$

Another pattern is $100\cdots00 - ve = 011\cdots10 + ve$. This pattern and its dual $(-1)00\cdots00 + ve$ cause a fine adjustment in the case of the previous work [64, 63]. The fine adjustment is basically to indicate that the location of the leading digit should be shifted by one position. We can mentally think of a first step detecting the leading zeros and the leading insignificant digits as being a coarse

detection while the fine adjustment is a following separate step. In the fine adjustment step if the leading digit is 1 and is not followed by another positive digit but by 0 then we must detect the sign of the remainder of the number. If that sign is negative, a fine adjustment is needed. The dual case holds for a leading -1.

The need to detect special cases for $+1$ followed by -15 or -1 followed by $+15$ can be eliminated by the use of some recoding techniques similar to what was presented in the work on recoders for partial compression [52, 53]. Daumas and Matula state [53]: "Partial compression also realizes virtually all the benefits of leading digit deletion." The word virtually is important; partial compression does not provide a solution for the fine adjustment cases. In fact, from a complexity and time delay point of view, getting the exact bit location of the leading one is essentially the same as doing a carry propagation. The fine adjustment is hence equivalent to transforming the redundant representation into a non-redundant one. As described by Quach and Flynn [64], parallel addition and leading one prediction are both problems of bit pattern detection. They also identified sticky bit computation as the third problem of bit pattern detection.

The problem of finding the exact location of the leading bit in signed digit number is also important for rounding. A rounding scheme for signed digit numbers was described by Matula and Nielsen [11] in an adder using the packet-forwarding paradigm [41]. In that work, they used a signed sticky digit as an indicator of the sign of the digit string after the first leading bit.

What is proposed in this work is to perform only the coarse adjustment of finding the leading digit by eliminating any leading zeros or insignificant digits. The main advantage of using a signed digit number system is to eliminate the carry propagation from the critical path. Replacing it with another circuit similar in complexity (fine adjustment) defeats the purpose. Hence, the fine adjustment is left to the rounding unit and a signed sticky digit is used there. The rounding occurs in parallel with the exponent difference and not sequentially after the addition, so it is out of the critical path. In fact, a comparison between the design presented here and the design of the adder using the packet-forwarding paradigm [41] reveals that they are similar but with crucial differences. In the proposed design, the final two to one adder of the packet-forwarding design is removed, the signed sticky calculation is lumped with the rounding (which uses signed digit addition as well) and both are done in parallel with the exponent difference of the following instruction. More details are presented in section 4.3.

Based on the work on partial recoders [52, 53], two recodings are defined to delete the leading insignificant digits. In the N-recoding a negative one is added to the digits and in the P-recoding a positive one is added. More specifically, for two consecutive digits of the result s_i and s_{i-1} with the MSB of s_{i-1} (s_{i-1_4}, the extra bit) having the same weight as the LSB of s_i,

$$\begin{array}{cccccccccc} \cdots & s_{i_3} & s_{i_2} & s_{i_1} & s_{i_0} & & s_{i-1_3} & s_{i-1_2} & \cdots \\ s_{i_4} & & & & s_{i-1_4} & & & & \end{array}$$

the N-recoding is defined as reseting s_{i-1_4} and s_{i_0} to 0 if they were both 1. This is mathematically correct since s_{i-1_4} has a negative value. This recoding can create digits that are equal to -16, however, this is not important since the recoded format is only used within the leading digit detector (LDD) circuits and even with such "out of bound" digits the position of the leading digit can be correctly estimated. The N-recoding when applied to the case of repeated -15 is:

digits	k	-15	$\cdots$	-15	l	$\cdots$
equiv.	$k_3k_2k_1k_0$	0001	$\cdots$	0001	$l_3l_2l_1l_0$	$\cdots$
bits	1	1	$\cdots 1$	l_4	$\cdots$	$\cdots$
result	$k-1$	0	$\cdots$	1	l	$\cdots$

If the digit k is equal to 1, then the insignificant pattern consisting of 1 followed by negative digits is eliminated by this N-recoding. The example assumes that the digit l is positive (i.e. $l_4 = 0$) giving a recoded result of $1\,l\,\cdots$. If l is negative then l_4 is canceled reducing the recoded result to $0\,(16+l)\,\cdots$. This feature is of value since it ensures that the number is truly normalized. A leading digit of 1 with a negative fractional part is not the normalized format. The condition for the N-recoding to change the bits is $s_{i_0} = s_{i-1_4} = 1$ and hence its output is given by $s^n_{i_0} = s_{i_0}\overline{s}_{i-1_4}$, $s^n_{i-1_4} = \overline{s}_{i_0}s_{i-1_4}$ while the remaining bits of the digit pass unchanged.

The P-recoding on the other hand is defined to eliminate the case of insignificant leading -1 followed by positive digits. Appendix A defines the P-recoding formally and discusses the order of implementation of the two recodings as well as some simplifications of the logic equations.

4.2.2 Second challenge: Rounding

As mentioned earlier, the fine adjustment is performed in the rounding stage. This is the other piece of challenging logic in the design at hand. In the proposed format the MSD has four bits. The rounding stage must determine the leading one among those four bits in order to decide on the approximate bit location for the rounding. The fine adjustment determines if the remaining part of the number below the leading bit of the MSD is positive or negative. Those two indicators allow for the decision on the correct bit location to apply the IEEE rounding.

Since the significand of the proposed format is always positive, the MSD has four bits only. Let us denote these bits by a (most significant), b, c and d (least significant). On the other hand, the LSD has five bits that we can denote by e (the extra bit with negative value), f (the most significant positive bit), g, h, i (the least significant). The digit containing the guard round and sticky digits is encoded so that $Guard = -2g_1 + g_0$ and $S = -2s_1 + s_0$. Hence it has g_1 (a bit from the guard digit with a negative value), g_0 (a bit from the guard digit with a positive value), r, s_1 (a bit from the sticky digit with a negative value) and s_0 (a positive value bit). The relative weight of each bit

Table 4.1: Rounding value for the four IEEE modes and different fractional ranges

range	*RNE*	*RZ*	*RP*		*RM*	
			$+ve$	$-ve$	$+ve$	$-ve$
$-1 < f < -0.5$	-1	-1	0	-1	-1	0
-0.5	$-L$	-1	0	-1	-1	0
$-0.5 < f < 0$	0	-1	0	-1	-1	0
0	0	0	0	0	0	0
$0 < f < 0.5$	0	0	1	0	0	1
0.5	L	0	1	0	0	1
$0.5 < f < 1$	1	0	1	0	0	1

is as follows:

a	b	c	d	$\cdots$	$\cdots$	$\cdots$	f	g	h	i	g_0	r	s_0
						e				g_1		s_1	

A fractional value f_i at bit location i of a signed digit binary number $\cdots x_{i+1}x_ix_{i-1}\cdots x_0$ can be defined as $f_i = (\Sigma_{j=0}^{j=i-1} 2^j \times x_j)/2^i$. The decision of the digit added for rounding is then determined by the fractional value at the rounding position. However, the value to add in order to achieve the correct rounding does not depend only on the fractional range but also on the IEEE rounding mode. In RP and RM modes, the sign of the floating point number affects the decision as well. The decision is according to Table 4.1 where L is the bit at the rounding location. Since the fractional value can be either positive or negative, the value added for rounding may be positive, negative or zero. Compared to conventional IEEE rounding logic, more complicated situations arise in some of the rounding cases for this redundant digit design. For example, the rounding to zero (RZ) mode of the IEEE is not just a simple truncation but a -1 is added to the number at the rounding location if the fractional value is negative.

In this design the fractional range is estimated and the rounding value decided speculatively for each bit location in the Least Significant Digit (LSD). The resulting potential new LSDs after adding each rounding value are also calculated. Then based on the circuits indicating the leading bit of the MSD and the fine adjustment, the final rounded LSD is chosen. The details of the logic equations governing the different cases and the circuit diagrams are given in appendix B.

4.3 Adder delay analysis and comparisons

The critical path of the proposed design starts with the exponent difference. This is a 15 bit adder and not an 11 bit one as in conventional adders using double precision because of the special format used in this design. In fact, the exponent width in this format $expWF$ is equal to the conventional exponent width $expW$ (which is dependent on n as specified by the IEEE standard) expanded to allow

for the normalization of denormalized numbers minus $\log_2 r$ when the radix is 2^r. The significand in this format is also larger than the corresponding significand for the conventional designs because of the redundancy. The significand width is $\lceil \frac{n}{r} \rceil \times (r+1) - 1$. The swapping multiplexers must be as wide as the significand and the output of the exponent difference is used to drive the select lines. Up to this point, the delay is estimated to be $5+2\times\lceil \log_{f-1}(\lceil \frac{expWF}{f} \rceil - 1)\rceil$ *FO4* delays for the exponent difference followed by $\lceil \log_4(\lceil \frac{n}{r} \rceil \times (r+1) - 1)\rceil + 1$ *FO4* delays for the swapping multiplexers. The operand then passes through a $\lceil \frac{n}{r} \rceil$-way shifter which adds $\lceil \log_2(\lceil \frac{n}{r} \rceil)\rceil$ *FO4* delays. The following multiplexer adds one more *FO4* delay. The signed digit adder takes $8 + 2 \times \lceil \log_{f-1}(\lceil \frac{r+1}{f} \rceil - 1)\rceil + \lceil \log_4(r+1)\rceil$ *FO4* delays. The select lines of the last multiplexer partially depend on the output of the adder in order to determine if there is a need to adjust to the right by one bit. Hence, there is a delay from the select lines to the output equal to $\lceil \log_4(\lceil \frac{n}{r} \rceil \times (r+1) - 1)\rceil + 1$ *FO4* delays. The total delay for this design is thus:

$$
\begin{aligned}
\tau_{add} \;=\; & 16 \\
& + 2 \times \lceil \log_{f-1}(\lceil \frac{expWF}{f} \rceil - 1)\rceil \\
& + 2 \times \lceil \log_4(\lceil \frac{n}{r} \rceil \times (r+1) - 1)\rceil \\
& + \lceil \log_2(\lceil \frac{n}{r} \rceil)\rceil \\
& + 2 \times \lceil \log_{f-1}(\lceil \frac{r+1}{f} \rceil - 1)\rceil + \lceil \log_4(r+1)\rceil
\end{aligned}
$$

From this derivation we can evaluate the benefit coming from each of the novel ideas in this design: the postponed rounding, the use of a higher radix base for the exponents and the use of an SD adder.

Once the significand addition is reduced by redundancy, the rounding delay must be masked by the delay of another part of the floating point operation that is as long or longer. Otherwise, it adds to the overall delay of the adder. The exponent difference and multiplexers time delay (second line and half of the third line of the equation) are both $\mathcal{O}(\log n)$ operations. Both are essential and are already on the critical path. Performing the rounding in parallel with them seems to be the best choice. Hence, the rounding delay is effectively hidden.

The higher radix base for the exponent has an effect on the alignment shifter which only shifts then to digit boundaries. The fourth line of the multi-line equation above captures this as a delay of $\lceil \log_2(\lceil \frac{n}{r} \rceil)\rceil$ rather than $\lceil \log_2(n)\rceil$ in conventional adders. Shifting is still an $\mathcal{O}(\log n)$ operation but its time delay is reduced by about $\log_2 r$ when using a higher radix.

The effect of the SD adder is shown in the fourth line (and part of the constant of the first line) where the terms $8+2\times\lceil \log_{f-1}(\lceil \frac{r+1}{f} \rceil - 1)\rceil + \lceil \log_4(r+1)\rceil$ appear instead of $5+2\times\lceil \log_{f-1}(\lceil \frac{n}{f} \rceil - 1)\rceil$ in a conventional adder.

The effect of r on the shifter delay is the opposite of its effect on the SD adder delay. The amount of redundancy (reflected by r) also has an effect on the area of the circuit and the required increase in the register file. Hence, depending on the choice of n and f for the implementation, the benefit from using a different radix for the exponent may be more than the benefit from the redundancy or vice versa. To compare this design to other designs some assumptions regarding those parameters are needed. For practical CMOS designs, the fan-in is usually limited to 3 or 4. The majority of the floating point adders are currently designed to handle double precision numbers ($n = 53$) or larger. For this range, the design proposed with r set to 4 or 8 provides the best performance [70] as presented in section 4.3.2 with more detail.

4.3.1 Conventional systems

A number of state of the art floating point adders [41, 66, 67, 71] are used for comparison based on the delay model described in chapter 3. As each of those published designs focuses on its own novel aspects, some assumptions are made when not enough details are presented in the other parts of the design. All of the designs selected use two-path algorithms for high-performance execution. In all those examples, the far path takes longer than the close path. The results reported here assume a longer far path.

Packet Forwarding adder [41]

The details of the packet forwarding adder are described in two other papers [11, 72]. The critical path for this adder goes through the exponent subtract (11 bit subtraction), the significand swap, and the alignment shifter (a full length shifter for the 65 bit significand) in the first cycle. In the second cycle, the critical path includes two $[4:2]$ compressors and the adjustment logic. The third cycle requires the sticky digit calculation (implemented by a tree of multiplexers and hence having $\lceil \log_2 n \rceil$ levels) and the rounding logic. The fourth cycle introduces the delay of the final operand width (64 bits) carry propagate adder. This design has a long latency compared to other designs but it provides an improved throughput due to the redundancy used. Using the model, the exponent subtract takes $5 + 2 \times \lceil \log_{f-1}(\lceil \frac{expW}{f} \rceil - 1) \rceil$ *FO4* delays. The significand swapping is done with a multiplexer that has the select lines coming from the exponent difference. So, $\lceil \log_4(n) \rceil + 1$ *FO4* delays is added for the swap. Next, the shifter takes $\lceil \log_2(n) \rceil$ *FO4* delays. The two $[4:2]$ compressors in the second cycle add 6 *FO4* delays. The adjust logic is not described in enough detail to make a good delay estimation. The adjust logic is based on the signs of the two numbers and the difference of the exponents, hence we can assume it to take only 2 *FO4* delays. The adjustment itself is a multiplexer whose select lines come from the adjust logic and takes $\lceil \log_4(n) \rceil + 1$ *FO4* delays. The signed sticky computation of the third cycle uses a tree of multiplexers [11] and hence takes $\lceil \log_2(n) \rceil$ *FO4* delays. The rounding logic is not specified and it has 10 inputs along with the mode (four possible modes) and the sign of the result. The rounding logic is then assumed to take

$\lceil \log_f(15) \rceil$ *FO4* delays. The final adder takes $5 + 2 \times \lceil \log_{f-1}(\lceil \frac{n}{f} \rceil - 1) \rceil$ *FO4* delays. To sum all of this, the packet forwarding adder has the following delay:

$$\begin{aligned} \tau_{pkt\ fwd} &= 20 + \lceil \log_f(15) \rceil + 2 \times \lceil \log_4(n) \rceil + 2 \times \lceil \log_2(n) \rceil \\ &\quad + 2 \times \lceil \log_{f-1}(\lceil \frac{expW}{f} \rceil - 1) \rceil + 2 \times \lceil \log_{f-1}(\lceil \frac{n}{f} \rceil - 1) \rceil \end{aligned}$$

As noted above, this design has a high latency of four cycles but provides a higher throughput by forwarding the operands after the second cycle. The time delay of the first two cycles determines the throughput and is given by:

$$\begin{aligned} \tau_{pkt\ fwd_{Thr}} &= 15 + 2 \times \lceil \log_4(n) \rceil + \lceil \log_2(n) \rceil \\ &\quad + 2 \times \lceil \log_{f-1}(\lceil \frac{expW}{f} \rceil - 1) \rceil \end{aligned}$$

Comparing with τ_{add} (the time for our proposed adder), we find that an addition instruction in our design saves its redundant result without rounding to the register file and is completed by the time τ_{add}. In the case of the packet forwarding design (*pkt fwd adder*), this is not true. It still goes through cycle 3 and 4 before committing its result to the register file. The addition is considered as a completed instruction only after cycle 4. Our design performs the rounding in parallel with the following instruction's exponent calculation while the *pkt fwd adder* does it sequentially after forwarding the redundant result. Obviously, a variation of the packet forwarding can be proposed where the redundant unrounded result is saved to the register file. Such a variation then does the sticky bit calculation and rounding somehow in parallel with the hardware of the first two cycles. That variation might be faster than our proposal but it suffers from the fact that the saved result is twice as large as the size of the IEEE numbers. Our design provides a solution to this by limiting the redundancy. The variation of the packet forwarding can be considered as a fully redundant design for floating point adders. This variation is included in the comparisons below to show the limiting case and is labeled as *pkt fwd*$_{Thr}$. The complete time delay of the *pkt fwd adder* is also included. The reader should notice that the complete time delay is quite large but the benefit is superior throughput.

Variable latency adder [66]

The variable latency adder (*VL adder*) has the exponent difference (11 bit) and the swap in its critical path for the first cycle. The second cycle includes the operand width shifter and the third cycle has the half adder, the carry propagate adder (operand width), and a multiplexer. The exponent subtract and the significand swap are on the critical path, similar to the case for the *pkt fwd adder*, and take $5 + 2 \times \lceil \log_{f-1}(\lceil \frac{expW}{f} \rceil - 1) \rceil$ and $\lceil \log_4(n) \rceil + 1$ *FO4* delays respectively. The shifter adds $\lceil \log_2(n) \rceil$ *FO4* delays. The half adder of the third cycle takes 2 *FO4* delays. Although not explicitly

reported, there must be a negation of the shifted operand in the case of a subtraction operation. This negation is assumed to take one *FO4* delay. The compound adder is assumed to take one *FO4* delay longer than a regular adder of the same width since it has an additional multiplexer to choose between the sum and the sum plus one. So the compound adder is assumed to take $6 + 2 \times \lceil \log_{f-1}(\lceil \frac{n}{f} \rceil - 1) \rceil$ *FO4* delays. The final multiplexer to choose between the far and close path adds one more *FO4* delay.

A few details that must exist and are not mentioned in the published work increase the estimation of the delay. Those are:

1. Post-normalization shifting: In case of a significand overflow there might be a need to shift the result by one bit location. The select lines of the multiplexer performing such a shift depend on the output of the adder. Hence, there is a delay from the select lines to the output equal to $\lceil \log_4(n) \rceil + 1$ *FO4* delays.

2. Comparing the exponent difference to the number of bits of the operands: If the difference is larger then the smaller operand is effectively reduced to zero and only affects the rounding. A delay of one multiplexer to choose an all zeros pattern as a second input to the adder might be required.

So, the variable latency adder has the following delay:

$$\begin{aligned} \tau_{VL} &= 18 + 2 \times \lceil \log_4(n) \rceil + \lceil \log_2(n) \rceil \\ &\quad + 2 \times \lceil \log_{f-1}(\lceil \frac{expW}{f} \rceil - 1) \rceil + 2 \times \lceil \log_{f-1}(\lceil \frac{n}{f} \rceil - 1) \rceil \end{aligned}$$

This is the longest path of the adder. The other shorter ones that are exploited to allow the variable latency feature are not taken into consideration for the comparison.

Reduced Latency adder [67]

The reduced latency adder (*RL adder*) has the exponent difference (11 bit), the swap and the operand width alignment shift in the first cycle. In the second cycle, the operand width carry propagate adder and some multiplexers fall on the critical path. Similar to the previous two designs, this design starts as well with the exponent difference and the significand swap which take $5+2\times\lceil \log_{f-1}(\lceil \frac{expW}{f} \rceil - 1) \rceil$ and $\lceil \log_4(n) \rceil + 1$ *FO4* delays respectively. The shifter adds $\lceil \log_2(n) \rceil$ *FO4* delays. The negation for the case of subtraction takes one *FO4* delay. The prefix adder produces several outputs that pass through logic to set some fields in the case of an exception (zero, NaN, ...). This is assumed to add one *FO4* delay to that of the adder giving $6 + 2 \times \lceil \log_{f-1}(\lceil \frac{n}{f} \rceil - 1) \rceil$ *FO4* delays. Then, there is a multiplexing stage to choose between the possible outcomes and a final multiplexer to choose between the far and near path. This is assumed to add two more *FO4* delays. The total delay for

the *RL adder* is thus:

$$\begin{aligned}\tau_{RL} &= 15 + \lceil \log_4(n) \rceil + \lceil \log_2(n) \rceil \\ &\quad + 2 \times \lceil \log_{f-1}(\lceil \frac{expW}{f} \rceil - 1) \rceil + 2 \times \lceil \log_{f-1}(\lceil \frac{n}{f} \rceil - 1) \rceil\end{aligned}$$

Fast IEEE FP adder [71]

The critical path of the fast FP adder (*fst adder*) has two possibilities. The first is the 7 bit exponent difference (not the full 11 bit), the bitwise XOR, the shifter and finally the multiplexer (from inputs to output). This first option takes $5 + 2 \times \lceil \log_{f-1}(\lceil \frac{7}{f} \rceil - 1) \rceil + 1 + \lceil \log_2(n) \rceil + 1$ *FO4* delays. The second possibility is the full 11 bit difference, the bitwise XOR, the OR tree, the delay from select lines to the output of the multiplexer. This latter option takes $5 + 2 \times \lceil \log_{f-1}(\lceil \frac{11}{f} \rceil - 1) \rceil + 1 + \lceil \log_f(5) \rceil + \lceil \log_4(n) \rceil + 1$ *FO4* delays. The second option is slightly longer and is the one used for the delay calculations if the exponent width is assumed to be 11 bits. Obviously, if the exponent width is only 8 bits then the first option (assuming that the whole exponent difference is evaluated by one adder) is the one determining the critical path. The second cycle of this design starts with the half adder and the compound adder which both add $7 + 2 \times \lceil \log_{f-1}(\lceil \frac{n}{f} \rceil - 1) \rceil$ *FO4* delays. The one bit location shifters are multiplexers whose select line is one of the bits produced by the compound adder, hence there is a delay of $\lceil \log_4(n) \rceil + 1$ *FO4* delays. The multiplexer used to choose between the two shifter outputs adds just one more *FO4* delay (the delay of its select line is in parallel with the delay of the select line of the previous shifters). Finally, there is a multiplexer to choose between the far and close path that adds one more *FO4* delay. To summarize, the *fst adder* has the following delay:

$$\begin{aligned}\tau_{fst,expW=11} &= 17 + \lceil \log_f(5) \rceil + 2 \times \lceil \log_4(n) \rceil \\ &\quad + 2 \times \lceil \log_{f-1}(\lceil \frac{11}{f} \rceil - 1) \rceil + 2 \times \lceil \log_{f-1}(\lceil \frac{n}{f} \rceil - 1) \rceil \\ \tau_{fst,expW=8} &= 17 + \lceil \log_4(n) \rceil \\ &\quad + 2 \times \lceil \log_{f-1}(\lceil \frac{8}{f} \rceil - 1) \rceil + 2 \times \lceil \log_{f-1}(\lceil \frac{n}{f} \rceil - 1) \rceil\end{aligned}$$

4.3.2 Comparison results

Since the significand width and exponent width are closely related in the formats used for floating point units, it is assumed here that the exponent width *expW* is eight bits for any significand width that is 24 bits or less. Otherwise, *expW* is assumed to be 11 bits. For the proposed design using the redundant digits, this translates to 11 bits with the small significand width and 15 bits otherwise. This difference in the exponent width leads to a sudden jump in the time estimated for the delay at the point where the significand width is 24. The assumption of such a step change is used instead

Table 4.2: Effect of r on the relative improvement of the adder for $n = 80$.

	$f = 3$		$f = 4$	
	$r = 4$	$r = 8$	$r = 4$	$r = 8$
RL adder	40		34	
Proposed	35	36	33	34
Relative improvement	$5/40 = 12.5\%$	10%	2.94%	0%

of a complex relation between the exponent width and significand width in order to simplify the derivation of the delay estimates. Another simplification used is to ignore small increases in the operand width (for example by one bit due to recoding in *design1*) along the critical path of the design.

Figure 4.3 shows the time delay of the different designs when the significand width varies from 8 to 120 bits for different values of fan-in from $f = 3$ to $f = 6$. For the design proposed in this thesis, r is kept constant at $r = 8$. When comparing all of the designs to the fully redundant "ideal" *pkt fwd*$_{Thr}$, it is clear that the redundancy in the adder proposed in this work makes it the fastest design for smaller fan-in values and large significand width. This is intuitively meaningful since for large significand widths the other designs suffer from a long time delay due to the longer carry propagation in the adders. In that same region of the design space, the *RL adder* seems to be the second best.

As the fan-in increases, the long carry delays can be made better by using larger groups of bits in the conditional-sum or carry-lookahead adders. Hence, the improvements in performance due to the redundancy become less important and the overhead due to the larger significand size make the design with redundancy less desirable as can be seen from the figure with $f = 6$. The overhead of the redundancy is also negating its benefits for the case of small n.

In the proposed design, considering the dependence of the exponent width on the radix and the other blocks (specially the shifter), the practical values for r should be multiples of 2. In order to minimize the additional cost of the redundancy (register storage, extra hardware,...) a large r is desirable. However, increasing r reduces the redundancy available and increases the time delay.

In Fig. 4.4, a comparison is presented between having $r = 4$ and $r = 8$ with the fan-in being either 3 or 4. Having a larger fan-in obviously improves the performance but it also decreases the relative advantage of this design compared to the other conventional ones. To clarify this point further, Table 4.2 compares the design proposed to that of the *RL adder* at $n = 80$ showing the relative improvement and the effect of r on the improvement.

To summarize, the region where the proposed design seems to outperform the conventional designs is when the fan-in is limited and the significand size is large (double precision and quad precision for example). Using more redundancy improves the speed for the larger significands but the price is a much larger area for the floating point adder and the register file. The limiting case

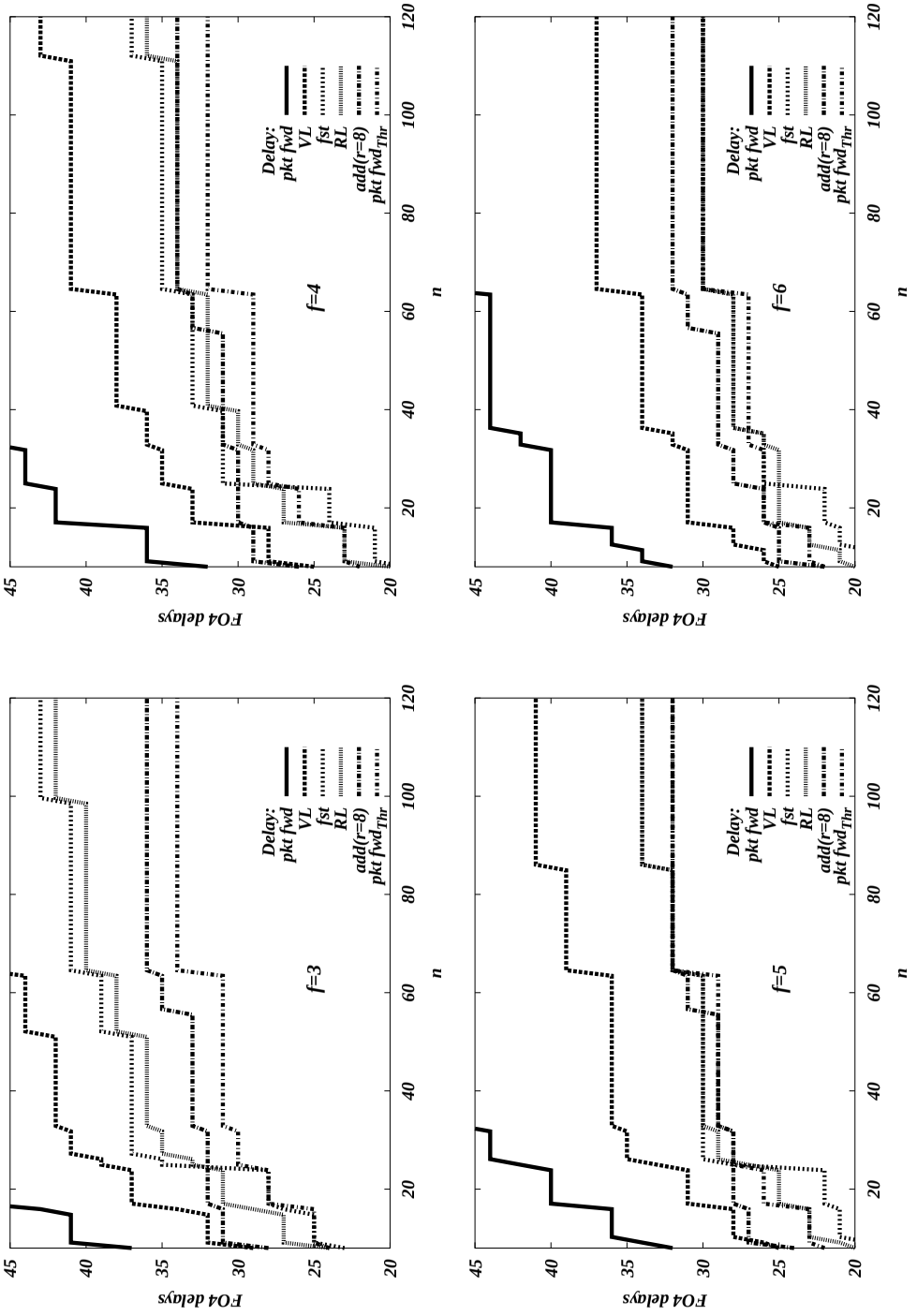


Figure 4.3: Time delay versus significand width for different fan-in values.

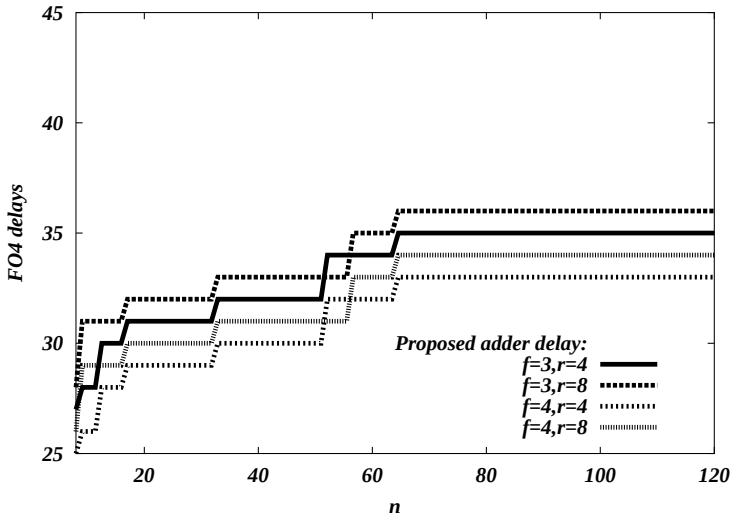


Figure 4.4: Comparison between $r = 4$ and $r = 8$ at $f = 3, 4$.

of increasing the redundancy is the "ideal" variant of the *pkt fwd adder* design.

4.4 Simulation results

The implementation details of the adder and the circuit schematics are provided in appendix C. From the schematics, a netlist of transistors representing the whole circuit as well as a complete Verilog description is extracted. Verilog is used to verify correct functionality. Exhaustive testing of the functionality is obviously not practical for such a design due to the large number of input bits. A testing Verilog module was written to generate random test vectors and input them to the adder. Manually checking the correctness of the output for random input test vectors proved to be a very labor intensive task. However, over a hundred such test vectors were manually checked to debug the complete design while exhaustive testing was done to the smaller components[1].

The transistor netlist is used for the timing simulation using the switch level simulator *irsim*. That timing simulation indicated the different gates and transistors that needed resizing to provide a strong enough source for the large fanouts that exist in different location of the design. Since the design is done in a scalable CMOS style, technology files ranging from $0.6\mu m$ down to $0.3\mu m$ are used. On that range of scaling factors, the adder performs as predicted by the analytical model when compared to the delay of *FO4* inverters at the same scaling factor. For the timing simulation of the adder, 5 000 test vectors as well as the assertions for the corresponding outputs were generated

[1] On the other hand, an automatic script was written to check the complete multiplier and it successfully passes more than 2.4 million random test vectors as discussed in chapter 5.

Table 4.3: Circuit statistics and simulation results for the floating point adder.

number of nodes	46 845
NMOS transistors	63 589
PMOS transistors	61 649
Test vectors	5 000
Model delay	$34FO4$
Simulation delay($0.6\mu m$)	$14ns = 33.35FO4$
Simulation delay($0.3\mu m$)	$6ns = 32.40FO4$

from the verilog simulator. Table 4.3 provides some of the circuit statistics as well as the simulation results.

4.5 Adder conclusions

An adder for the redundant digit floating point system was proposed in this chapter. The two main challenges appearing in this design were the leading digit detection and the rounding. Both of those blocks were discussed and the possible alternatives in their design explored.

The comparison of the proposed adder with other work indicates that the proposed adder is more suited for the large significand sizes and small fan-in technology. To verify the assertions made about the speed advantage, a circuit with a specific significand width (corresponding to the double precision), as well as a specific fan-in limitation and redundancy was implemented at the transistor level. The simulation results of that implementation confirm the prediction from the simpler analytical model.

Chapter 5

Multiplication unit

5.1 Conventional Multiplier designs and their time delays

Hand multiplication is a serial task where a digit of one operand (the multiplier) is multiplied by the digits forming the other operand (the multiplicand) and the result is a partial product that is written down before going to the next digit of the multiplier. The partial products are formed in this serial fashion and are summed to deliver the final result. On the other hand, high speed multiplier parallelize the task of generating the partial products by using extra hardware to handle all the digits of the multiplier simultaneously. In addition, the summation of the partial products to form the last result is broken into two steps: parallel reduction of the partial products to two numbers and then the final summation of those two numbers.

In the case of binary numbers, the generation of the partial products is done easily using *AND* gates. Such a simple scheme generates as many partial products as the number of bits in the multiplier. To decrease the time it takes to sum them, Booth recoding of the multiplier is usually employed to reduce the number of partial products generated. When more than two bits are used for the recoding, some hard multiples are needed. The redundant Booth-3 [73] is a possible solution for this problem. In this solution, redundancy is used inside the multiplier circuits while the operands to the unit remain non-redundant. The Booth recoding adds some delay in the generation and reduces the delay of the summation. Whether this is beneficial overall or not is a difficult question to answer. It could be with no benefit at all in some cases as some researchers claim [74]. While others assert that it saves between 11% and 50% of the delay [75]. A recent empirical study [76] indicated that the answer depends on a variety of factors including the operand sizes and the type of topology used to sum the partial products as well as the technology of implementation.

Partial product reduction in high speed multipliers is important as it is the piece of the multiplier that consumes the most time and area. Hence different approaches have been used to optimize it.

1. The reduction of the partial products is usually done by the use of an internal redundant representation for the numbers.

   (a) Even if the original operands are not redundant, three of the partial products are summed to give a carry vector and a sum vector. This is a redundant representation of the sum and constitutes a reduction of three product vectors to two (carry and sum) vectors. Those latter two are combined with other vectors and the reduction ratio of three to two continues. The reduction occurs by using a number of logic elements in parallel with each reducing three input bits to two output bits. Each of those elements is called a $(3, 2)$ counter which is simply a full adder circuit that takes two inputs plus a carry to generate a sum and a carry out bit. An alternative logic element is the $[4 : 2]$ compressor [77, 12] which adds four input bits plus a carry to produce two output bits and a carry and achieves a reduction ratio of two to one.

   (b) Other schemes of reduction use signed digits [35, 39, 34, 37].

2. The optimum placement and routing of the counters to reduce the partial products to the final sum and carry was studied by several people [60, 78, 79]. The empirical study of Al-Twaijry [76, 80] indicates the region of superiority for each of the topologies for partial product reduction given the operand width and the technology used.

The final carry propagate adder in the high speed multipliers is characterized by the fact that not all its inputs arrive at the same time and hence it is also optimized in a variety of ways [60]. However, it still accounts for almost 30% of the delay of the multiplier [73]. In the case of producing a redundant output as in the system proposed in this work, this final adder is replaced by a signed digit adder that is faster and simpler to design.

5.2 Proposed Multiplier design

Any design for a parallel floating point multiplier involves a number of functions to be performed. Namely, unpacking the operands into the exponent and the significand parts, partial products generation using the significands, reduction of the partial products, final addition, normalization (if needed) then packing the result back into the floating point format. Similar to the adder, in the proposed system, the multiplier does not round the numbers at the end of the multiplication operation but rather at the start of the following operation. So, the rounding in this system can occur either in the unpacking stage or in the partial product generation stage. These two choices are not similar and some analysis is needed to determine the best approach. Another subtle difference between the multiplier in the proposed system and the conventional multipliers is the fact that each operand has a number of bits that are considered to have a negative value. These are the extra bits in each digit of the number. The significands of the two operands X and Y are thus assumed to

have three components: P the positively valued bits, E the negatively valued extra bits and r the value of the rounding digit which is either positive or negative (i.e. $r \in \{-1, 0, 1\}$.) The end result is $X \times Y$ but, $X = P_x - E_x + r_x$ and $Y = P_y - E_y + r_y$ and hence a number of possibilities for the design is studied. Multiplying those two operands results in some partial products which are positive and some which are negative. The two's complement of each negative partial product can be added to form the final result. Another approach is to have all these negative partial products added separately first then complemented and summed with the positive ones. The generation of the partial products can be done in different ways by breaking the operands into their sub-parts and handling each one separately. This gives the following options (neglecting the rounding issue for a moment)

1. $X \times Y$ directly
2. $P_xY - E_xY$
3. $P_xY - E_xP_y + E_xE_y$
4. $P_xP_y - P_xE_y - E_xP_y + E_xE_y$

If Y is used as a direct multiplier (options 1, 2, 3) then a modified Booth recoding must be used because of the extra bits which have a negative value. The involved logic recodes Y in a different signed digit set that is minimally redundant. For radix-2^k Booth (Booth k for short), the produced digits are in the set $\{-2^k/2, \cdots, 2^k/2\}$. Since Y is originally represented by a redundant set as well, this is a case of conversion between two redundant sets. Such conversions are proven to take a fixed amount of time independent on the operand size [29]. The specific case of Booth recoding of signed binary (where each bit is signed) has been studied previously [48]. It was proven that each digit in Booth k recoding is determined by $2k + 1$ consecutive input signed bits. In the system proposed here, only $2k$ bit locations are needed and the reason is that Y has mainly unsigned positive bits. It is only the bit locations where the extra bits occur that are considered as "signed bits." For Booth 2 or 3 a truth table can be derived giving the output in each case. The fact that Booth recoding produces correct results for multiplication has been proved in different ways in the research literature [49, 47].

Each of the problems specific to the design at hand is explained in the following sections before introducing the delay analysis and the simulation results.

5.2.1 First challenge: "Negative" bits and Booth recoding

Among the different implementation options for the multiplier mentioned above, option 4 requires no extra modified recoders but the multiplication of P_xE_y involves a full width (but reduced height) partial products array (PPA). This is saved by adding the recoders needed for options 2 and 3. This seems to be a large hardware saving. There might be more latency if the recoders are on the critical

path but still this is to some extent compensated by the fact that option 4 has more terms to add up at the end which takes more time.

In a direct implementation of option 2 we have both positive and "negative" valued bits in the E_xY calculation. If we generate the partial products and add the positive separately from the negative, this reduces to option 3.

Option 1 is the most desirable since it is the most compact. However, it introduces "negative" valued bits into the PPA. Those can be summed separately and then subtracted from the positive bits. Another solution is to use a scheme with bias similar to the one used in redundant Booth 3. In a biased Booth system [73], the bias constant K has most of its bits with a value of 0 except for a few with 1. In the redundant Booth 3, it has 1 in the locations where the carry bits occur. So if, for example, 5 bit adders are used, the carries occur every 5 bits and the constant is $K = \cdots 01000010000100000$. The partially redundant form (after the addition of the bias constant) has the additional bits at the location that is one bit higher than the locations of the bias constant. Similarly, if a biased Booth scheme is used with the proposed system, then only one carry bit is allowed as redundancy for each group of bits. This requirement means that the bias constant has 1 in the locations where the extra bits occur. Obviously, the location of the extra bit is not the same in the multiple, X, and its double, $2X$. Thus, the bias constant has 1 at the highest location of the extra bit and a short addition occurs to find out the partially redundant form as shown in Table 5.1 for a Booth 2 system. In this table, e is the extra bit and it has a negative value as indicated by the minus sign. The positive bits of X are a, b and c. For each case, the boolean expressions used to drive the bits x, y and z of the result after adding the constant K are given. Other bit groups have similar logic in them.

However, the interval between the extra bits is 4 which is a multiple of 2. This leads to the accumulation of the additional bits of the partially redundant form at certain columns in the PPA. A solution is to use Booth 3 instead with the interval remaining 4. In such a case, the constant has 1 in a location one bit higher than the example of the Booth 2 above to accommodate the $3X$ multiple. It is necessary to add X and $2X$ to generate $3X$ and use additional gates to include the constant. This design involves more delay in generating the $K \pm 3X$ multiples and can become complicated for wiring.

So, this idea is not further considered. The proposed solution is to generate two bit vectors for each partial product (PP). One of those is treated as a positive vector and the other as negative. The positive vectors are summed together and the negative ones are summed together. At the end, the difference between the two sums is calculated. If a Booth 2 recoding is used with this solution, there is no need to worry about sign extension. To generate $-X$, one of two possibilities can be used. The first is to select E_x for the positive vector and P_x for the negative vector. The second possibility is to negate X using SD addition in parallel with the recoding of Y by the Booth recoders. The negative of X may be called mX as a short notation for minus X and is composed of P_{mx} and E_{mx}

Table 5.1: Biased Booth 2.

$K+X$	$\cdots a$	b	c	$\cdots$	$x = e \oplus c$
		1	$-e$		$y = b \oplus (\overline{e} + c)$
	$\cdots a$	y	x	$\cdots$	$z = b(\overline{e} + c)$
	z				
$K-X$	$\cdots \overline{a}$	$\overline{b}$	$\overline{c}$	$\cdots$	$x = e \oplus \overline{c}$
		1	e		$y = b \oplus (e\overline{c})$
	$\cdots \overline{a}$	y	x	$\cdots$	$z = \overline{b} + e\overline{c}$
	z				
$K+2X$	$\cdots b$	c	d	$\cdots$	$x = d$
		$1-e$			$y = c \oplus \overline{e}$
	$\cdots b$	y	x	$\cdots$	$z = c\overline{e}$
	z				
$K-2X$	$\cdots \overline{b}$	$\overline{c}$	$\overline{d}$	$\cdots$	$x = \overline{d}$
		$1+e$			$y = c \oplus e$
	$\cdots b$	y	x	$\cdots$	$z = \overline{c} + e$
	z				

which is used when $-X$ is needed. The double of the multiple is simply a one bit left shifted version of the multiple and similarly for the negative of the double (shifted version of $-X$). There is no need for any sign extension if one of these two possibilities is implemented. The second possibility (producing mX) is what is used in the implementation presented here.

At the end of the partial products reduction, the negative sum is subtracted from the positive sum. The combination of those last four bit vectors can be done in two ways:

1. The two positive PP's and the two negative PP's are assumed as forming only two numbers with signed digits. Each signed digit is represented by two bits one from a positive vector and one from a negative vector. Those two signed numbers can be summed and the result produced in the proposed format.

2. The negative PP's are complemented and a row of $[4:2]$ compressors is used to reduce the four PP's to only two. Then these last two are used to produce the proposed format.

In the first possibility, each 4 bits range from -15 (all ones in the negative PP and all zeros in the positive PP) to $+15$ (the opposite case). So, they are in the same range as the digits in the proposed format but they are in a different encoding. A special adder can be designed to add these two signed numbers using SD addition rules and produce the result in the required format.

Basically, each signed binary position has a primary sum in $\{-2, -1, 0, 1, 2\}$. However, the final sum bit can only be in $\{0, 1\}$ except for the MSB of the sum (the sign bit of the two's complement)

which can be in $\{0, -1\}$. A system of internal negative carries—alternately named: borrows—is thus required to fulfill this function. If the primary sum of a position is already -2 and a borrow is needed from it then its value becomes -3. This -3 is handled by generating a sum of 1 and a carry of -1 to the position that is two bits higher (i.e. $-3 = -4 + 1$). That position that is two bits higher could then have two carries coming to it. One from the position directly below it and another from the position two bits lower. This is the maximum number of carries to any position.

In the second possibility, each bit of the negative PP's is complemented. A constant equal to 2 must be added in the reduction tree to make this complementation equivalent to a two's complement. Then those two are added to the positive PP's using $[4 : 2]$ compressors. The result is two PP's in two's complement form. It is important to note that when the negative PP's are complemented, they are signed extended by one bit. The corresponding bits in the positive PP's are filled with 0's. The resulting PP's after the $[4 : 2]$ compression can be either negative or positive. However, they cannot be both negative since the final result must be positive (both of the significands being multiplied are positive). The two numbers are divided into groups of 4 bits to be added as if they are SD numbers. In all the digits, the design of the SD adder implemented in the floating point adder can be used. A factor that simplifies its design here even further is that all its input digits are positive except for the MSD which might be negative. So, in all the digit locations, $(x_i + y_i)$ and $(x_i + y_i + 1)$ are pre-calculated together with $c_{i(1)}$. There is no need to calculate $c_{i(-1)}$ or $(x_i + y_i - 1)$ since a negative carry is never produced. The same is true for the MSD location (where negative digits can occur) since the end result must be positive.

From a delay point of view, the two possibilities seem to be comparable. However, from an ease of implementation and simplicity point of view, the second possibility appears to be superior and was selected for the multiplier in Fig. 5.1.

Booth recoding

A few multiplier designs deal with the issue of having a redundant input operand [33]. As noted above, in those designs a need arises for modifying the Booth recoders. With Booth recoding, the extra bit (which has a negative value) can fall under either of the bits being considered by the recoder. This bit has then a value of $0, -1, -2, \cdots$. Each k bit locations in a Booth k system is recoded as having a new value and a "carry out." Those two are based on the original value, the extra bit, and the "carry in." Obviously, the carry out is dependent only on the group of bits being considered —together with any extra bits it may contain— but independent of the carry into that group. Otherwise, there is a carry propagation problem and the conversion does not occur in parallel.

Tables 5.2 and 5.3 indicate the possible recoding schemes for Booth 2 and Booth 3. Similar tables can be developed for other Booth systems. The column with the extra bit equal to zero is the same as the conventional Booth recoders. It is important to note that these tables are not unique but

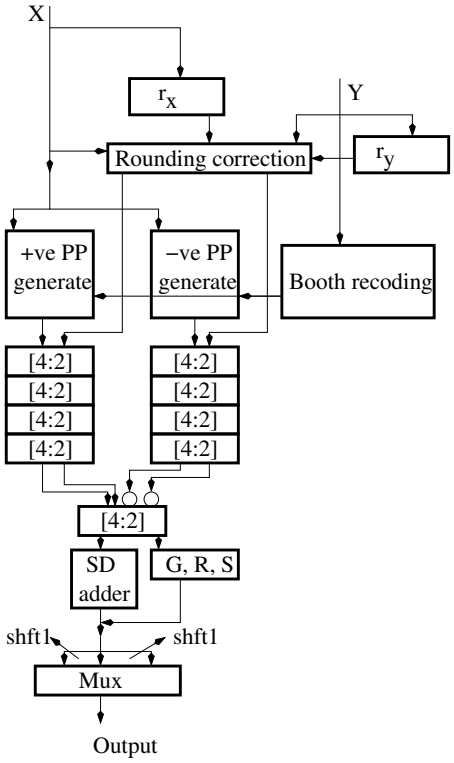


Figure 5.1: Conceptual block diagram of the multiplier.

Table 5.2: Booth 2 recoding with the extra bit.

			extra = 0		extra = -1		extra = -2	
b_2	b_1	c_i	v	c_o	v	c_o	v	c_o
0	0	0	0	0	-1	0	-2	0
0	0	1	1	0	-0	0	-1	0
0	1	0	1	0	0	0	-1	0
0	1	1	2	0	1	0	-0	0
1	0	0	-2	1	1	0	0	0
1	0	1	-1	1	2	0	1	0
1	1	0	-1	1	-2	1	1	0
1	1	1	-0	1	-1	1	2	0

other possibilities for the recoding exist. As an example, for the last line in Table 5.2 and the case of extra bit equal to -2, the value can be recoded as $v = -2$ and $c_o = 1$. However, the tables where chosen to have the least amount of carry propagation possible. This also gives a useful property, namely, the column giving v is rotated by two times the amount of the extra bit when compared to conventional recoders. This helps a designer to check the correctness of the table derived for any Booth k.

Obviously in the system at hand, if Booth 3 is to be used, it must be some sort of a redundant Booth 3. Otherwise, the delay and area of the adder used to get the precise three times multiple overshadows the benefits gained from eliminating the carry propagation adder at the end. The chosen option for the multiplication (regardless of the Booth recoding) is already redundant. It has each multiple represented by two vectors, one positive and one negative. If the $3M$ multiple required for Booth 3 is made available in such a form, then the multiplication is easily done. In fact, $3X = 2X + X = 2P_x + P_x - (2E_x + E_x)$. Since E_x is a sparse vector with a gap of three 0's between each two bits, the $2E_x + E_x = 3E_x$ part can be formed just by repeating the extra bits in one location higher. A row of full adders is used to add P_x, $2P_x$ (a shifted version) and the complement of the vector $3E_x$. The bit by bit complement of $3E_x$ is only the one's complement but it must be made two's complement. Hence a 1 is added in the LSB (empty location) of the $2P_X$ vector. The row of full adders compresses the three vectors into two. Since X is known to be positive, then $3X$ must be positive as well. Among the three vectors that are added, two are positive and one is negative. For compatibility with the rest of the multiplier, one of the resulting two vectors should be positive and the other negative. The following Lemma proves that this is indeed the case.

Lemma 5.2.1 *If a row of full adders is used to sum three binary vectors represented in two's complement form with any two vectors being non-negative (positive or zero) while the third vector is stricly negative then the result is two vectors having opposite signs.*

Proof: Let us assume that the two non-negative vectors are labeled P and Q and that they have

Table 5.3: Booth 3 recoding with the extra bit.

				extra = 0		extra = -1		extra = -2		extra = -4	
b_4	b_2	b_1	c_i	v	c_o	v	c_o	v	c_o	v	c_o
0	0	0	0	0	0	-1	0	-2	0	-4	0
0	0	0	1	1	0	-0	0	-1	0	-3	0
0	0	1	0	1	0	0	0	-1	0	-3	0
0	0	1	1	2	0	1	0	-0	0	-2	0
0	1	0	0	2	0	1	0	0	0	-2	0
0	1	0	1	3	0	2	0	1	0	-1	0
0	1	1	0	3	0	2	0	1	0	-1	0
0	1	1	1	4	0	3	0	2	0	-0	0
1	0	0	0	-4	1	3	0	2	0	0	0
1	0	0	1	-3	1	4	0	3	0	1	0
1	0	1	0	-3	1	-4	1	3	0	1	0
1	0	1	1	-2	1	-3	1	4	0	2	0
1	1	0	0	-2	1	-3	1	-4	1	2	0
1	1	0	1	-1	1	-2	1	-3	1	3	0
1	1	1	0	-1	1	-2	1	-3	1	3	0
1	1	1	1	-0	1	-1	1	-2	1	4	0

n bits with the MSB $P_{n-1} = Q_{n-1} = 0$. Similarly the negative vector is R with $R_{n-1} = 1$. The results are in two's complement form given by S the vector of the sum bits generated from the full adders and T the vector of the carry bits. S has the same number of bits as the added vectors and T is one bit longer with $S_i = P_i \oplus Q_i \oplus R_i$, $T_{i+1} = P_iQ_i + P_iR_i + Q_iR_i$ and $T_0 = 0$. From these relations, it is clear that $S_{n-1} = 1$ and $T_n = 0$ which means that S is negative and T is positive.$\diamond$

In the system discussed here, P_x extended by a 0 at the MSB side becomes P, $2P_x$ padded with a 1 at the LSB side becomes Q and the one's complement of $3E_x$ becomes R. Since $-3E_x \leq 0$ then if R is interpreted as representing a two's complement number as in the Lemma it is indeed a strictly negative number. The conditions for the Lemma are thus satisfied. The resulting positive vector is used directly. The resulting negative vector, however, requires a small change to suit the multiplier scheme discussed above. The second vector in each partial product has an absolute value which is considered to be indicating a negative number. Hence, two's complementation is needed to get the absolute value of the negative vector resulting from the full adders. This can be achieved by bit complementation and the 1 needed to make it two's complement is inserted in the following partial product as is done in conventional multipliers with Booth recoding.

Other ways for preparing the $3X$ multiple can be devised as well.[1] However, the modified recoders for a Booth 3 system are larger and slower than those of the Booth 2 as can be derived

[1] For example, using SD addition rules to add X and $2P_x$ and then insert $2E_x$ in the negative vector. This can be even done in parallel with the rounding step.

from Tables 5.2 and 5.3. It was also shown [76] that, in general, Booth 2 is the recoding scheme which minimizes the latency of a multiplier when technology scaling is taken into account. For this reason a Booth 2 recoding is used in the design presented here.

For the multiplier operand Y, Table 5.2 is used to implement a Booth 2 recoding scheme. Since the gap between the extra bits is 4, then in a Booth 2 scheme the extra bits always fall under the bit b_1 of Table 5.2. Let us denote it by e_1. The possible values for the extra bit are thus only 0 or -1 and the boolean expression giving the possible outputs become:

$$\begin{aligned}
(0) &= \overline{e}_1\overline{b}_2\overline{b}_1\overline{c}_i + e_1\overline{b}_2 b_1\overline{c}_i \\
(1) &= \overline{e}_1\overline{b}_2(b_1 \oplus c_i) + e_1(\overline{b}_2 b_1 c_i + b_2\overline{b}_1\overline{c}_i) \\
(2) &= \overline{e}_1\overline{b}_2 b_1 c_i + e_1 b_2\overline{b}_1 c_i \\
(-0) &= \overline{e}_1 b_2 b_1 c_i + e_1\overline{b}_2\overline{b}_1 c_i \\
(-1) &= \overline{e}_1 b_2(b_1 \oplus c_i) + e_1(\overline{b}_2\overline{b}_1\overline{c}_i + b_2 b_1 c_i) \\
(-2) &= \overline{e}_1 b_2\overline{b}_1\overline{c}_i + e_1 b_2 b_1\overline{c}_i \\
c_o &= b_2(\overline{e}_1 + b_1)
\end{aligned}$$

It is important to note that in $2Y$, the extra bits always fall under b_2. Since the last column of Table 5.2 has always 0 as a value for c_o, then boolean expressions derived in such a case could be simpler. In fact, if the extra bit always falls under b_2 (with a value of either 0 or -2) and is denoted by e_2 then the boolean expressions become:

$$\begin{aligned}
(0) &= \overline{e}_2\overline{b}_2\overline{b}_1\overline{c}_i + e_2 b_2\overline{b}_1\overline{c}_i \\
(1) &= \overline{e}_2\overline{b}_2(b_1 \oplus c_i) + e_2 b_2(b_1 \oplus c_i) \\
(2) &= \overline{e}_2\overline{b}_2 b_1 c_i + e_2 b_2 b_1 c_i \\
(-0) &= \overline{e}_2 b_2 b_1 c_i + e_2\overline{b}_2 b_1 c_i \\
(-1) &= \overline{e}_2 b_2(b_1 \oplus c_i) + e_2\overline{b}_2(b_1 \oplus c_i) \\
(-2) &= \overline{e}_2 b_2\overline{b}_1\overline{c}_i + e_2\overline{b}_2\overline{b}_1\overline{c}_i \\
c_o &= \overline{e}_2 b_2
\end{aligned}$$

These expressions are indeed simpler. In order to use them the multiplication is assumed to be $(X \times 2Y)/2$. The doubling of Y is a simple left shift by one. The result of $X \times 2Y$ must end with a zero as its least significant bit. This bit is dropped and the rest of the result taken to achieve the division by two needed at the end. Another way to look at this is to assume that the Booth recoding

is done to the operand Y when its digits are divided as:

$$\cdots p_{y_7} \;\Big|\; p_{y_6} \quad p_{y_5} \;\Big|\; \begin{matrix} p_{y_4} \\ e_{y_0} \end{matrix} \quad p_{y_3} \;\Big|\; p_{y_2} \quad p_{y_1} \;\Big|\; p_{y_0}$$

with the low order bit not recoded but multiplied separately. This assumption is what is implemented and, in fact, it is more than just the LSB that is treated separately. The whole least significant digit is handled in a special manner due to rounding.

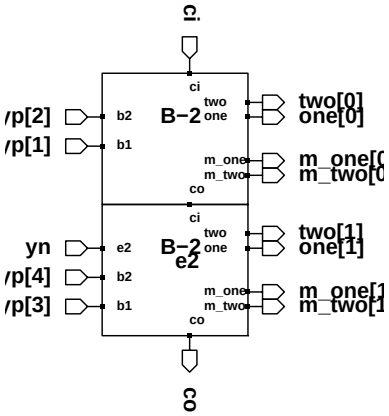
Fig. 5.2 shows the schematic of the circuits used for Booth 2 recoding. The generation of c_o in the cell dealing with the extra bit takes two gate delays and then this signal is fed into the following regular Booth recoder as its c_i. That c_i affects the outputs of the recoder after two more gate delays. Hence, the longest path in the Booth recoding scheme presented is estimated to be 4 *FO4* gate delays.

5.2.2 Second challenge: When to round

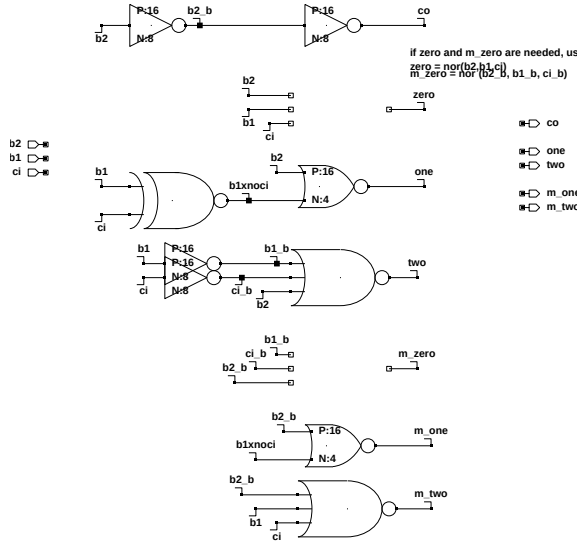
Similar to the case of the floating point adder, rounding in the presented floating point multiplier proved to be a challenge. The operands can be rounded first and then the rounded significands used to perform the multiplication. This puts the rounding logic on the critical path of the multiplier and delays the start of the partial products reduction. A second possibility is to start the multiplication of the $(P_x - E_x)$ and $(P_y - E_y)$ parts while, simultaneously, generating additional partial products with the values $r_y(P_x - E_x)$, $r_x(P_y - E_y)$ and $r_x r_y$. In this second case, the reduction of the partial products starts directly but it takes more hardware because of the added partial products.

Using the parametric delay model of chapter 3, we can evaluate both possibilities. In the first possibility, the rounding is done first before generating the partial products. The delay for this is estimated to be $\lceil \log_f(r) \rceil + \lceil \log_f(\lceil n/r \rceil) \rceil$ for the determination of the signed sticky digit (to generate the inputs of and get through with Fig. B.2) followed by 2 *FO4* delays for the multiplexers selecting the carries out of the least significant digit (see Fig. B.1) and then a delay of $\lceil \log_4(r+1) \rceil + 1$ for the multiplexer selecting the next higher digit. With $n = 54$, $f = 3$ and $r = 4$ the time delay for rounding in this case is estimated to be 10 *FO4*. In comparison, if a tree of $[4:2]$ compressors is used for the partial products reduction then it has $(\lceil \log_2(\lceil n/r \rceil) \rceil - 1)$ levels with each taking 3 *FO4* gate delays. That tree when $n = 54$ gives a delay of 12 *FO4* gate delays. Obviously, rounding first without any other parallel work done on the significands is unacceptable if a high speed multiplier is required.

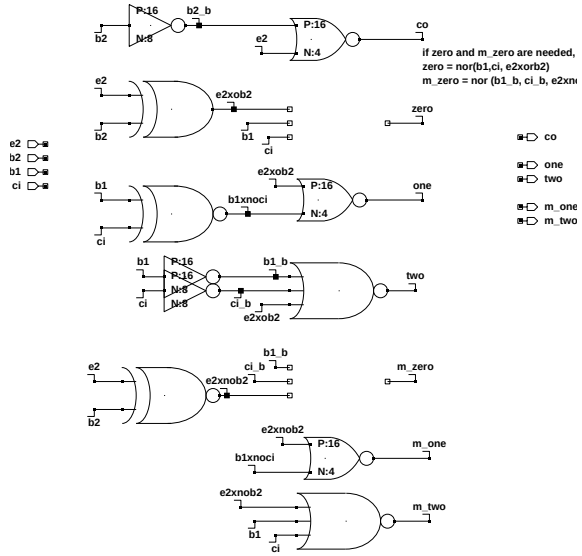
The other possibility is then to chop both X and Y at the rounding location. As mentioned in the discussion on rounding in section 4.2.2, this location is determined by the leading bit of the MSD and the signed sticky digit. The operand chopping takes away all the bits below the approximate rounding location determined *only* by the leading bit of the MSD. The multiplication is carried as $(X_{chopped} + (b_x + r_x))(Y_{ch} + (b_y + r_y))$ where b_x and b_y are the bits directly after the rounding location



The lower cell deals with the extra bit while the upper is a regular Booth recoder.



The regular Booth recoder.



The recoder handling the extra bit.

Figure 5.2: The building block of the Booth 2 recoding.

in case the signed sticky indicates a negative fractional value while r_x and r_y are the rounding digits. Hence the multiplication can be divided into three parts:

$X \times Y =$	$X_{chopped}Y_{chopped}$	By Booth recoding, generating partial products and summing
	$+(b_x + r_x)Y_{chopped} + (b_y + r_y)X_{chopped}$	2 special partial products added in the tree
	$+(b_x + r_x)(b_y + r_y)$	special correction added to the tree

The derivation of the logic equations for those corrections as well as the circuits implementing them are presented in appendix B.

5.3 Multiplier delay analysis and comparisons

Two different paths affect the generation of the partial products: the preparation of mX (the negative of X) and the recoding of Y. Once the pre-processing ends (Y is recoded, X is chopped and mX is available), the generation of each partial product is done via a row of multiplexers whose select lines are the recoded Y digits. The number of multiplexers in each row is approximately $\lceil \frac{n}{r} \rceil \times (r+2)$. Although the width of the operands is estimated to be $\lceil \frac{n}{r} \rceil \times (r+1) - 1$ the estimation of the number of multiplexers in the each row uses $(r+2)$ and not $(r+1)$ since in the case of choosing the double of X the extra bit is shifted by one location. The partial products are then reduced by a tree of $[4:2]$ compressors. Because of the Booth recoding, the number of partial products is $\lceil \frac{n}{2} \rceil$. As discussed in the rounding section above, three correction vectors are added to the partial products. The number of levels in the tree to reduce the product to two bit vectors is estimated as $\lceil \log_2(\lceil \frac{n}{2} \rceil + 3) \rceil - 1$. Following that is the compressor combining the positive and negative sum then the SD adder.

It takes 4 *FO4* gate delays to negate a digit as shown in Fig. 5.3. Both X and mX must be buffered to supply the $\lceil n/2 \rceil$ rows of multiplexers. The buffering delay is thus $\lceil \log_4 \lceil n/2 \rceil \rceil$. Hence the total delay to generate and buffer mX is estimated to be $4 + \lceil \log_4 \lceil n/2 \rceil \rceil$. One more gate delay is needed to get the output of the partial product multiplexer which puts the total delay according to this path starting with X at $5 + \lceil \log_4 \lceil \frac{n}{2} \rceil \rceil$.

The delay of the Booth recoder is 4 *FO4* gate delays as mentioned in section 5.2.1. The outputs of the recoders are used as select lines for the row of $\lceil \frac{n}{r} \rceil \times (r+2)$ multiplexers and the time delay to the output is then $\lceil \log_4(\lceil \frac{n}{r} \rceil \times (r+2)) \rceil + 1$. The total time delay to get the output of the partial product multiplexer according to this path is thus $5 + \lceil \log_4(\lceil \frac{n}{r} \rceil \times (r+2)) \rceil$ which is larger than the path from X described above.

Each of the compressors in the tree as well as the one after the tree takes 3 *FO4* gate delays resulting in $3(\lceil \log_2(\lceil \frac{n}{2} \rceil + 3) \rceil - 1) + 3 = 3\lceil \log_2(\lceil \frac{n}{2} \rceil + 3) \rceil$ *FO4* gate delays. The SD adder used here is much simpler than the one used in the floating point adder. In fact, it is similar to a number of regular adders in parallel each of them dealing with only $(r+1)$ bits. Hence, its time delay

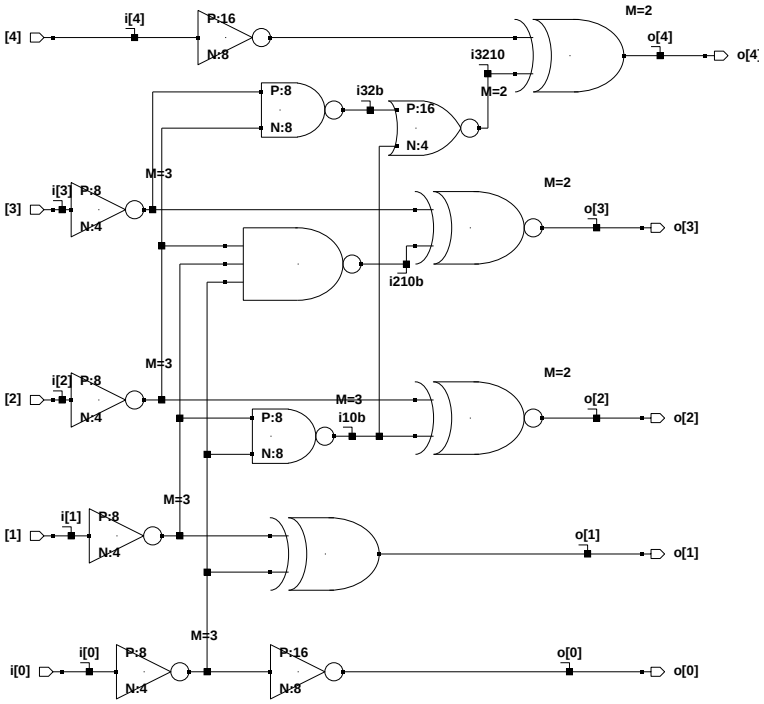


Figure 5.3: Producing the negative of one digit.

is the same as a regular adder with just one extra *FO4* delay added to it to accommodate for choosing between the sum and sum plus one. The time delay of that adder is then estimated to be $6 + 2 \times \lceil \log_{f-1}(\lceil \frac{r+1}{f} \rceil - 1) \rceil$. The select lines of the normalization shifter at the end depend on the outputs from the adder and hence it adds an estimated delay of $\lceil \log_4(\lceil \frac{n}{r} \rceil \times (r+1) - 1) \rceil + 1$. The total delay of the multiplier is then estimated to be:

$$\begin{aligned} \tau_{mul} &= 12 \\ &\quad + \lceil \log_4(\lceil \frac{n}{r} \rceil \times (r+2)) \rceil \\ &\quad + 3\lceil \log_2(\lceil \frac{n}{2} \rceil + 3) \rceil \\ &\quad + 2 \times \lceil \log_{f-1}(\lceil \frac{r+1}{f} \rceil - 1) \rceil \\ &\quad + \lceil \log_4(\lceil \frac{n}{r} \rceil \times (r+1) - 1) \rceil \end{aligned}$$

The effect of the fan-in limit f of the gates is minimal in this design. It only appears in the third line of the equation above. In fact, for the practical values of $f = 3, 4$ and $r = 4, 8$, the third line evaluates to zero and the change of f has no effect.

5.3.1 Conventional systems

Conventional systems vary widely in their use of Booth recoding and in the topology used to reduce the partial products. As the recent empirical study [76] showed these choices can have a big effect on the time delay and area. As a comparison, a conventional system using Booth 2 and a tree of $[4:2]$ compressors with a final carry propagate adder is assumed here.

As shown in Fig. 5.2, the regular Booth recoding takes 2 *FO4* gate delays. To drive the partial product multiplexers and get to their output, $\lceil \log_4(n) \rceil + 1$ *FO4* gate delays are needed. Similar to above, the compressors in the tree take 3 gate delays each resulting in $3(\lceil \log_2 \lceil \frac{n}{2} \rceil \rceil - 1)$ *FO4* gate delays. Since the multiplier produces a result that is double the size of the operand but in the case of the floating point numbers only the most significant half is kept, the final adder is assumed to be only n bits wide. The delay of that adder is then $5 + 2 \times \lceil \log_{f-1}(\lceil \frac{n}{f} \rceil - 1) \rceil$ *FO4* gate delays. The normalization multiplexer adds another $\lceil \log_4(n) \rceil + 1$ *FO4* gate delays. The total time delay for a conventional multiplier is then:

$$\begin{aligned} \tau_{con} &= 6 + \lceil \log_4(n) \rceil + 3(\lceil \log_2 \lceil \frac{n}{2} \rceil \rceil) \\ &\quad + 2 \times \lceil \log_{f-1}(\lceil \frac{n}{f} \rceil - 1) \rceil + \lceil \log_4(n) \rceil \end{aligned}$$

Table 5.4: Effect of r on the relative improvement of the multiplier for $n = 80$.

	$f = 3$		$f = 4$	
	$r = 4$	$r = 8$	$r = 4$	$r = 8$
Conventional	42		38	
Proposed	38	40	38	40
Relative improvement	$4/42 = 9.5\%$	4.8%	0%	-5.3%

5.3.2 Comparison results

Fig. 5.4 illustrates the comparison between the multiplier proposed in this work and the conventional multipliers. As indicated above, when f is increased the proposed design does not benefit from it while the conventional design is speeded up since the final carry propagate adder gets faster.

The sudden steps in the plots are due to the ceiling function used in the estimation of the time delays.

The same general conclusions regarding the floating point adder apply here as well. The proposed design is good when the significand width is large. As the fan-in increases, the proposed design loses its attractiveness. Smaller r indicates higher redundancy which decreases the time delay. Table 5.4 compares the design proposed to a conventional one at $n = 80$ showing the relative improvement and the effect of r on the improvement.

When comparing the time delay equations, we find that the improvement brought by the use of the faster SD adder are offset to some extent by the slower Booth recoder and the need to sum the negative and positive partial products using an extra $[4 : 2]$ compressor.

5.4 Simulation results

Similar to the floating point adder, a netlist of transistors representing the whole multiplier circuit as well as a complete Verilog description are extracted from the schematics. Those schematics and the implementation details are presented in appendix D.

An automatic script was written to check the Verilog description of the multiplier. The script generates a clock signal driving two modules, one for generating the stimulus of the multiplier and one for checking its outputs. The stimulus generation module generates two random inputs as operands. Those inputs are checked to make sure that operands fit the required conditions:

- No digit should have an extra negative bit of 1 while the corresponding positive bits being all zero. This insures that no digit of value -16 is used.
- If the MSD is randomly selected to be zero it is set to 1 to insure that the numbers are normalized.

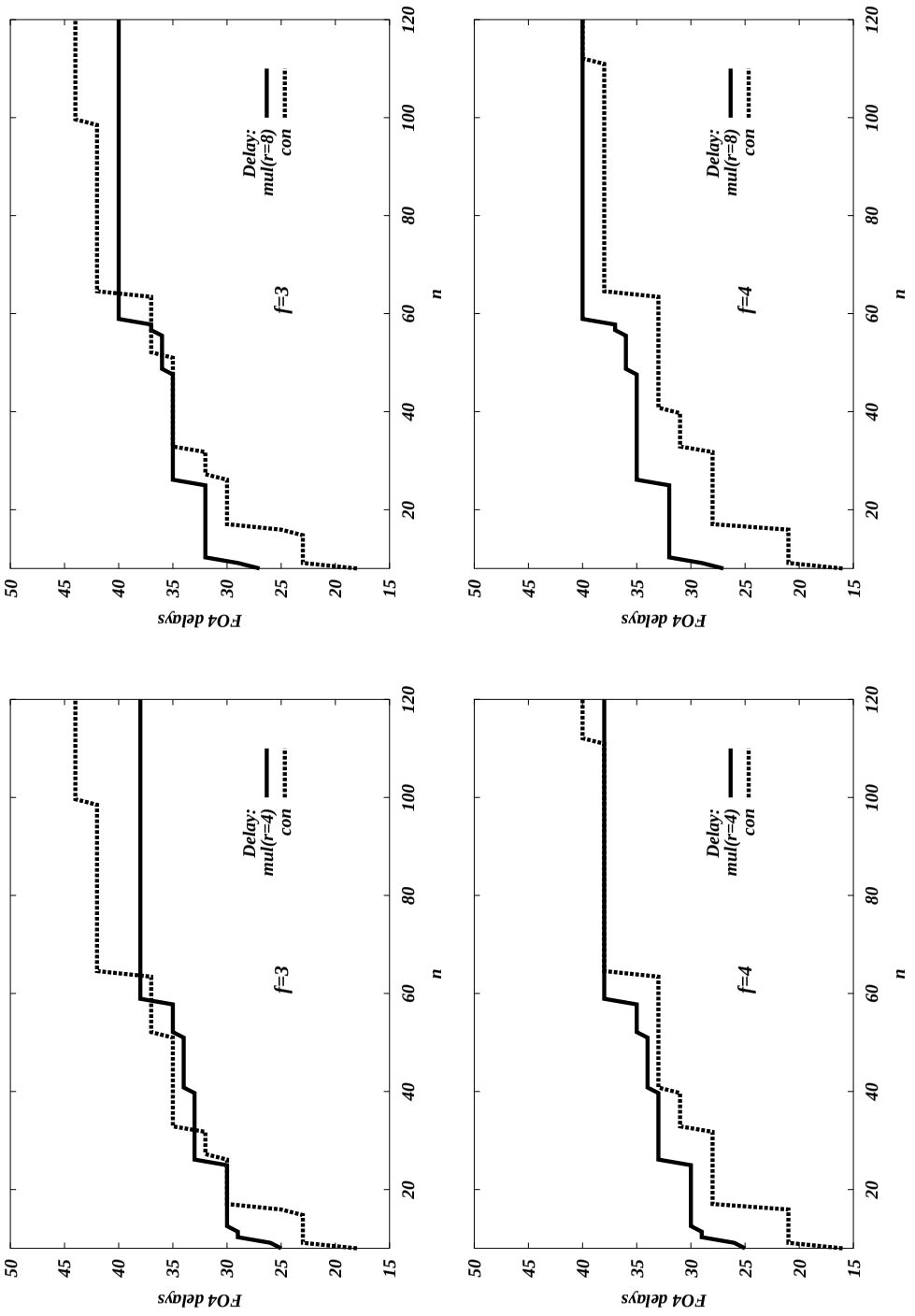


Figure 5.4: Multiplier time delay versus significand width for different f and r values.

- The design assumes that the MSD is similar to any other digit and has an extra negative bit. This must always be set to zero to insure that the MSD is positive as required by the format of the proposed system. The Verilog module generating the inputs does just that.
- if the MSD is equal to 1, the adjacent lower digit must be positive. Hence, its extra bit is set to zero in the case of an MSD equal to 1.
- In the digit for the guard, round and sticky, if the random number is chosen so that the guard digit is equal to -2 and the round bit is equal to 0 then the sticky digit is set to 01 to insure that the fractional range at the least significant bit of the LSD is bounded between -1 and 1 as mentioned in appendix B.

Once the rounding mode and the sign of each operand are chosen all these signals are supplied to the floating point multiplier.

The outputs of the multiplier are checked by the checking module. That checking module starts by calculating the rounded non-redundant representation of both operands and multiplying them together. That product is then checked for the different possibilities of normalization and is shifted accordingly and truncated at the rounding position to give a value named *XtimesY-trunc*. The sticky digit is calculated from the truncated part and is padded to *XtimesY-trunc*. At the same time, the non-redundant (but not rounded) representation of the output from the multiplier is also calculated. This representation is then compared to the *XtimesY-trunc*. If they match the result is correct.

This checking mechanism as well as similar ones for the smaller components enabled the discovery of several mistakes along the way of the design. The checking for the complete multiplier was left to run for a couple of weeks and the multiplier successfully passes more than 2.4 million random test vectors. Although this number constitutes a very small portion of the test vector space, it still gives a level of confidence in the algorithm and implementation. Since a formal proof of correctness is beyond the scope of the current work the focus shifted to the speed once this confidence was achieved.

Similar to the floating point adder, the timing simulation helped to indicate the gates and transistors that needed resizing. On the range of scaling factors from $0.6\mu m$ down to $0.3\mu m$ the multiplier performs as predicted by the analytical model when compared to the delay of *FO4* inverters at the same scaling factor. The Verilog simulator was used to generate the test vectors for the timing simulation. In total, 10 000 test vectors as well as the assertions for the corresponding outputs were generated. The simulation results as well as some of the circuit statistics are given in Table 5.5.

Table 5.5: Circuit statistics and simulation results for the floating point multiplier.

number of nodes	76 523
NMOS transistors	104 695
PMOS transistors	105 037
Test vectors	10 000
Model delay	$35FO4$
Simulation delay($0.6\mu m$)	$14.8ns = 35.25FO4$
Simulation delay($0.3\mu m$)	$6.4ns = 34.60FO4$

5.5 Multiplier conclusions

The multiplier proposed in this chapter together with the floating point adder presented in the previous chapter constitute the fundamental blocks for a floating point unit using the proposed redundant format.

The comparison of the proposed multiplier with the conventional multiplier is consistant with the adder comparison. The design is better suited for the large significand sizes and technologies where the fan-in is limited. The relative gain from the proposed multiplier over conventional multipliers is not as large as the relative gain in the case of the adder.

To verify the assertions made about the speed advantage, a circuit with a specific significand width (corresponding to the double precision), as well as a specific fan-in limitation and redundancy was implemented. The simulation results confirm the prediction based on the simpler analytical model.

Chapter 6

Division and elementary functions unit

6.1 Conventional Divider designs and their time delays

The integer division operation is defined as the function resulting in the quotient Q and remainder R from the dividend N and divisor D using the following equation:

$$N = QD + R$$

This is one equation having two unknowns and would yield an infinite number of solutions if there is no other condition set. The conditions chosen usually involve the magnitude of Q and the range of R. A few examples are: [17]

1. Modulus division is when $0 \leq R < D$
2. Signed division is when the magnitude of Q is independent of the signs of N and D.
3. Floor division is when Q is the greatest integer (note that $-3 > -4$) that is contained by N/D.

On the other hand, for floating point numbers, the IEEE standard separates between the definitions of the division and the remainder operation. For division, the standard defines the sign of the quotient as the exclusive or of the operands' signs. The quotient is defined as the rounded version of the infinitely precise result. Depending on the rounding mode, the quotient may be more or less than the infinitely precise result.

On the other hand, the remainder is defined regardless of the rounding mode by the mathematical relation $R = N - D \times n$ where n is the integer nearest to the exact value of N/D. If $\mid n - N/D \mid = 1/2$

then n is chosen to be even. Thus the remainder is always exact. If $R = 0$, its sign is defined to be that of N.

Floating point division is accomplished in hardware by an algorithm belonging to one of three general classes: table lookup, subtractive or multiplicative [27, 9, 1].

Table lookup: A direct table lookup method might use the dividend and divisor bits as address lines for a ROM storing the values of the quotient. Obviously, this is only feasible for very short significands. If each exceeds 10 bits or so, the table size becomes intolerable. A simple variation is to use the lookup table to find the reciprocal of the divisor and then multiply it by the dividend. That variation allows for larger significands but the size of the table becomes a limit quickly since it grows exponentially with the size of its input. More variations result in indirect table lookup methods where there is some processing of the numbers before and after the table lookup. The interpolation table method and the bipartite table method fall in this category. The advantage of table lookup is their speed since there is almost no computation time needed and the indirect methods make table lookup techniques feasible for IEEE single precision numbers. However, the table size is prohibitively large for longer significands. Table lookups can still be used for double and quad precision numbers as a starting approximation that is refined by another method (usually a multiplicative technique.)

Subtractive: This class is also called digit recurrence algorithms because the techniques used are iterative based on digit addition, subtraction and shifting. It includes the restoring and non-restoring division, SRT division, the CORDIC algorithm, continued fractions algorithms and very high radix techniques. This class of algorithms produces a remainder as a by product of the quotient computation. Hence, exact rounding is easy to implement. Since these techniques iterate on the digits of the divisor, the convergence to the required quotient precision is linear. With the exception of the very high radix techniques, this class is only suited for short precision operations because of the linear conversion. If used with large significands it produces a slow result only acceptable in low or medium performance hardware. The very high radix techniques can be used with large significands but they are not a match in speed to the multiplicative algorithms.

Multiplicative: Those include polynomial approximation, rational approximation, the division by repeated multiplication and the division by reciprocation. The reciprocation techniques can be further divided into series expansion and Newton-Raphson method. The convergence of this class is at least quadratic with the number of bits of precision doubling every iteration. If a higher order version of an algorithm such as the Newton-Raphson is used the convergence is even faster. This class is the best suited for high speed dividers dealing with large significands. The disadvantage of this class is the fact that the remainder is not produced as a by product and exact rounding is a bit more complicated.

6.2 Conventional elementary functions units and their time delays

Elementary functions are the exponential, logarithm, trigonometric, inverse trigonometric, hyperbolic and inverse hyperbolic functions. For the sake of this work, we add to this group the square root function.

These are all functions of one operand. Table lookup methods are thus attractive if the precision of the operand is small. For larger operand sizes, the CORDIC algorithm has been used extensively for evaluating the elementary functions in scientific calculators and medium performance arithmetic coprocessors. Since CORDIC is a linearly converging slow algorithm, it does not lend itself to high speed hardware. For that latter category, polynomial and rational approximations can be used as well as series expansion and Newton-Raphson methods.

6.3 Proposed Divider and elementary functions unit

To perform division and other elementary functions, a design from the literature is adapted [1, 2]. This arithmetic unit (shown in Fig. 6.1) provides rapid convergence based on higher-order Newton-Raphson and series expansion techniques.

Division is done by finding the reciprocal first and then multiplying the dividend by the reciprocal of the divisor. When using a fourth-order Newton-Raphson iteration, the reciprocal of a number b is given by

$$\frac{1}{b} = x_0(1 + d + d^2 + d^3 + d^4)$$

where x_0 is the approximate reciprocal found by a small lookup table and $d = (1 - bx_0)$. If the number of bits in b is n and the order of the iteration is k (which is four in the equation above) then the $\frac{n}{k+1}$ most significant bits of b can be used to access the lookup table and produce x_0. This approximation of the reciprocal is approximately $\frac{n}{k+1}$ bits wide and is then used to calculate $(1 - bx_0)$ as shown in the figure. The number of bits used to access the table and the number of bits produced by the table can be optimized to reduce the table size without sacrificing the required reciprocal error bound for exact rounding [1]. The arithmetic unit achieves fast computation by using parallel squaring, cubing, and powering units. These units compute the higher-order terms significantly faster than the traditional approach of serial multipliers. In effect, these powering units are even simpler than regular multipliers and produce their results faster [81]. All of the terms are computed in parallel and their results are left in carry save form further reducing the latency.

This same architecture can be used to compute the square root and inverse square root if two more lookup tables are provided to produce the approximations $y_0 = 1/\sqrt{b}$ and $z_0 = \sqrt{b}$. Then,

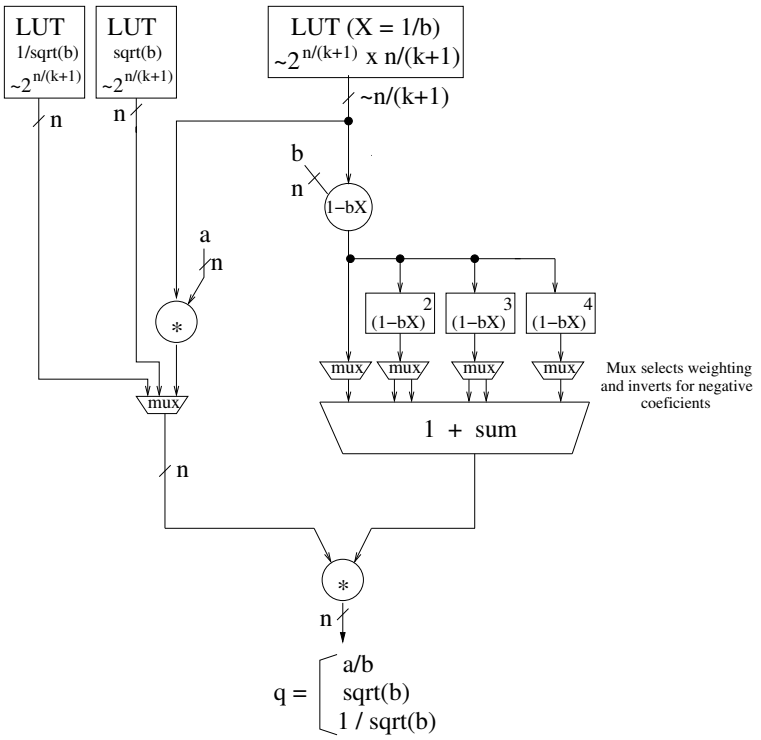


Figure 6.1: Division and elementary functions unit [1, 2].

using the same d, $\sqrt{b}$ and $\frac{1}{\sqrt{b}}$ are calculated by the equations:

$$\begin{aligned}\sqrt{b} &= y_0(1-\frac{1}{2}d-\frac{1}{8}d^2-\frac{1}{16}d^3-\frac{15}{128}d^4)\\ \frac{1}{\sqrt{b}} &= z_0(1+\frac{1}{2}d+\frac{3}{8}d^2+\frac{5}{16}d^3+\frac{35}{128}d^4)\end{aligned}$$

The outputs of the powering units are kept in carry-save form and pass through a row of multiplexers to select the correct coefficient depending on the function implemented. This selection of the coefficient is done by appropriately shifting and adding the corresponding power of d without requiring any further multiplications. All the powers are then summed together and multiplied by either ax_0 to give $\frac{a}{b}$ or y_0 to give $\sqrt{b}$ or z_0 to give $\frac{1}{\sqrt{b}}$. If other lookup tables are provided more functions can be evaluated.

This architecture can be pipelined and hence it provides a fast and efficient function evaluation while allowing high-throughput.

To adapt this architecture to the format proposed here, a short adder is used to eliminate the redundancy from the most significant part of the divisor operand by subtracting the extra bits. This non-redundant part is used to access the lookup table while the rest of the operand is fully transformed into a non-redundant form. In parallel, another adder is used to convert the dividend into a non-redundant form as well. The unit then works on those two operands as in the original design. To enhance the speed, the sum of the $(1-bx_0)^i$ terms is kept in carry save form. Finally, in the last multiplier, a signed digit adder is used instead of the regular carry propagate adder.

6.4 Delay analysis and comparison

The critical path of the adapted unit starts with an adder eliminating the redundancy of the most significant $\frac{n}{k+1}$ bits. This adder has a delay of $5+2\times\lceil\log_{f-1}(\lceil\frac{n}{f(k+1)}\rceil-1)\rceil$ *FO4* gate delays. A lookup table with L input lines has a time delay of [17]:

$$\text{Lookup table delay} = 2+\lceil\log_f(\frac{L}{2})\rceil+\lceil\log_f(2^{\frac{L}{2}})\rceil$$

Hence, if $\frac{n}{k+1}$ bits are used to access the table the time delay associated with it is $2+\lceil\log_f(\frac{n}{2(k+1)})\rceil+\lceil\log_f(2^{\frac{n}{2(k+1)}})\rceil$. The result of the table is also $\frac{n}{k+1}$ bits wide. This first approximation x_0 is used in a small multiplier to generate $(1-bx_0)$.

Since x_0 has a small number of bits, no Booth recoding is used but rather the partial products are generated directly using *AND* gates then a tree of $[4:2]$ compressors is used to reduce them to two bit vectors. The tree has $(\lceil\log_2(\frac{n}{k+1})\rceil-1)$ levels. A carry propagate adder is used to add the two bit vectors and generate $(1-bx_0)$. That adder has a delay of $5+2\times\lceil\log_{f-1}(\lceil\frac{n}{f}\rceil-1)\rceil$ *FO4*

gate delays. So far, the delay of the unit is:

$$\begin{aligned}
&5+2+1+5+2\times\lceil\log_{f-1}(\lceil\frac{n}{f(k+1)}\rceil-1)\rceil\\
&+\lceil\log_f(\frac{n}{2(k+1)})\rceil+\lceil\log_f(2^{\frac{n}{2(k+1)}})\rceil\\
&+3\times(\lceil\log_2(\frac{n}{k+1})\rceil-1)+2\times\lceil\log_{f-1}(\lceil\frac{n}{f}\rceil-1)\rceil
\end{aligned}$$

According to Liddicoat [1], the squaring unit for a number with n bits has $(\lfloor\frac{n}{2}\rfloor+1)$ partial products. The time delay for the squaring unit is thus equal to one *FO4* gate to generate the partial products and $3\times(\lceil\log_2(\lfloor\frac{n}{2}\rfloor+1)\rceil-1)$ *FO4* gate delays to reduce them. Simultaneously, the cubing and higher order units produce their results. The multiplexers selecting the weights add one more *FO4* delay. A tree of $[4:2]$ compressors is used to sum and reduce the results of the powering units to two bit vectors. There is at most $2k-1$ vectors since the original $(1-bx_0)$ term is not in carry save form. Hence the summing tree has $(\lceil\log_2(2k-1)\rceil-1)$ levels.

The result of the branch calculating ax_0 is forwarded to a Booth recoder in the last multiplier. Less constraints exist on this path and the output of the Booth recoder is ready before the time the summation of the powering units results is done. The Booth 2 recoding reduces the number of partial products to $\lceil\frac{n}{2}\rceil$. This compensates for the full redundancy (carry save form) of the sum of the powering units. Hence the actual number of partial products is estimated to be n. It takes one *FO4* gate delay to select the appropriate partial product depending on the Booth recoding output then reduction tree is composed of $(\lceil\log_2(n)\rceil-1)$ levels. Finally, the signed digit adder increases the delay by $8+2\times\lceil\log_{f-1}(\lceil\frac{r+1}{f}\rceil-1)\rceil+\lceil\log_4(r+1)\rceil$. The total time delay of the unit is:

$$\begin{aligned}
\tau_{div,elem} &= 13+2\times\lceil\log_{f-1}(\lceil\frac{n}{f(k+1)}\rceil-1)\rceil\\
&\quad+\lceil\log_f(\frac{n}{2(k+1)})\rceil+\lceil\log_f(2^{\frac{n}{2(k+1)}})\rceil\\
&\quad+3\times(\lceil\log_2(\frac{n}{k+1})\rceil-1)+2\times\lceil\log_{f-1}(\lceil\frac{n}{f}\rceil-1)\rceil\\
&\quad+1+3\times(\lceil\log_2(\lfloor\frac{n}{2}\rfloor+1)\rceil-1)\\
&\quad+1+3\times(\lceil\log_2(2k-1)\rceil-1)\\
&\quad+1+3\times(\lceil\log_2(n)\rceil-1)\\
&\quad+8+2\times\lceil\log_{f-1}(\lceil\frac{r+1}{f}\rceil-1)\rceil+\lceil\log_4(r+1)\rceil\\
\tau_{div,elem} &= 12+\lceil\log_f(\frac{n}{2(k+1)})\rceil+\lceil\log_f(2^{\frac{n}{2(k+1)}})\rceil+2\times\lceil\log_{f-1}(\lceil\frac{n}{f}\rceil-1)\rceil\\
&\quad+3\times(\lceil\log_2(\frac{n}{k+1})\rceil+\lceil\log_2(\lfloor\frac{n}{2}\rfloor+1)\rceil+\lceil\log_2(2k-1)\rceil+\lceil\log_2(n)\rceil)\\
&\quad+2\times(\lceil\log_{f-1}(\lceil\frac{n}{f(k+1)}\rceil-1)\rceil+\lceil\log_{f-1}(\lceil\frac{r+1}{f}\rceil-1)\rceil)+\lceil\log_4(r+1)\rceil
\end{aligned}$$

Table 6.1: Effect of r on the relative improvement of the division and elementary functions unit for $n = 80$.

	$f=3$		$f=4$	
	$r=4$	$r=8$	$r=4$	$r=8$
Original	101		87	
Proposed	95	97	85	87
Relative improvement	$6/101 = 5.94\%$	3.96%	2.3%	0%

The original unit does not have the extra adder that is used to eliminate the redundancy at the start. It has a comple carry propagate adder to sum the outputs of the powering units and hence the number of the partial products in the final multiplier is half what it is in our proposed adaptation. Finally, the last multiplier of the original unit has a carry propagate adder and not a signed digit adder. Hence, the delay of the original unit is:

$$\begin{aligned}
\tau_{orig} &= \tau_{div,elem} \\
&\quad - (13 + 2 \times (\lceil \log_{f-1}(\lceil \frac{n}{f(k+1)} \rceil - 1) \rceil + \lceil \log_{f-1}(\lceil \frac{r+1}{f} \rceil - 1) \rceil) + \lceil \log_4(r+1) \rceil) \\
&\quad - 3 \times \lceil \log_2(n) \rceil \\
&\quad + (2 \times (5 + 2 \times \lceil \log_{f-1}(\lceil \frac{n}{f} \rceil - 1) \rceil) + 3 \times \lceil \log_2(\frac{n}{2}) \rceil) \\
\tau_{orig} &= \tau_{div,elem} - 3 + 3 \times (\lceil \log_2(\frac{n}{2}) \rceil - \lceil \log_2(n) \rceil) - \lceil \log_4(r+1) \rceil \\
&\quad + 2 \times (2 \times \lceil \log_{f-1}(\lceil \frac{n}{f} \rceil - 1) \rceil - \lceil \log_{f-1}(\lceil \frac{n}{f(k+1)} \rceil - 1) \rceil - \lceil \log_{f-1}(\lceil \frac{r+1}{f} \rceil - 1) \rceil)
\end{aligned}$$

Fig. 6.2 shows a comparison between the proposed adaptation and the original unit when $k = 4$. Similar to the case of the adder and multiplier, our adaptation is better when the fan-in is limited. A higher redundancy (smaller r) improves the performance further. Table 6.1 compares the design proposed to a conventional one at $n = 80$ showing the relative improvement and the effect of r on the improvement.

The improvement in the case of the division and elementary functions unit is even smaller than the case of the adder or multiplier. For practical reasons, the adapted unit and the original one can be assumed to have the same time delay.

6.5 Divider and elementary functions unit conclusions

For high speed and large significands, the multiplicative or series expansion approach is the most suitable for division and elementary functions. A previously proposed unit capable of performing high speed, high throughput division and elementary functions is adapted to the proposed format. This adaptation is practically taking the same time delay of the original unit.

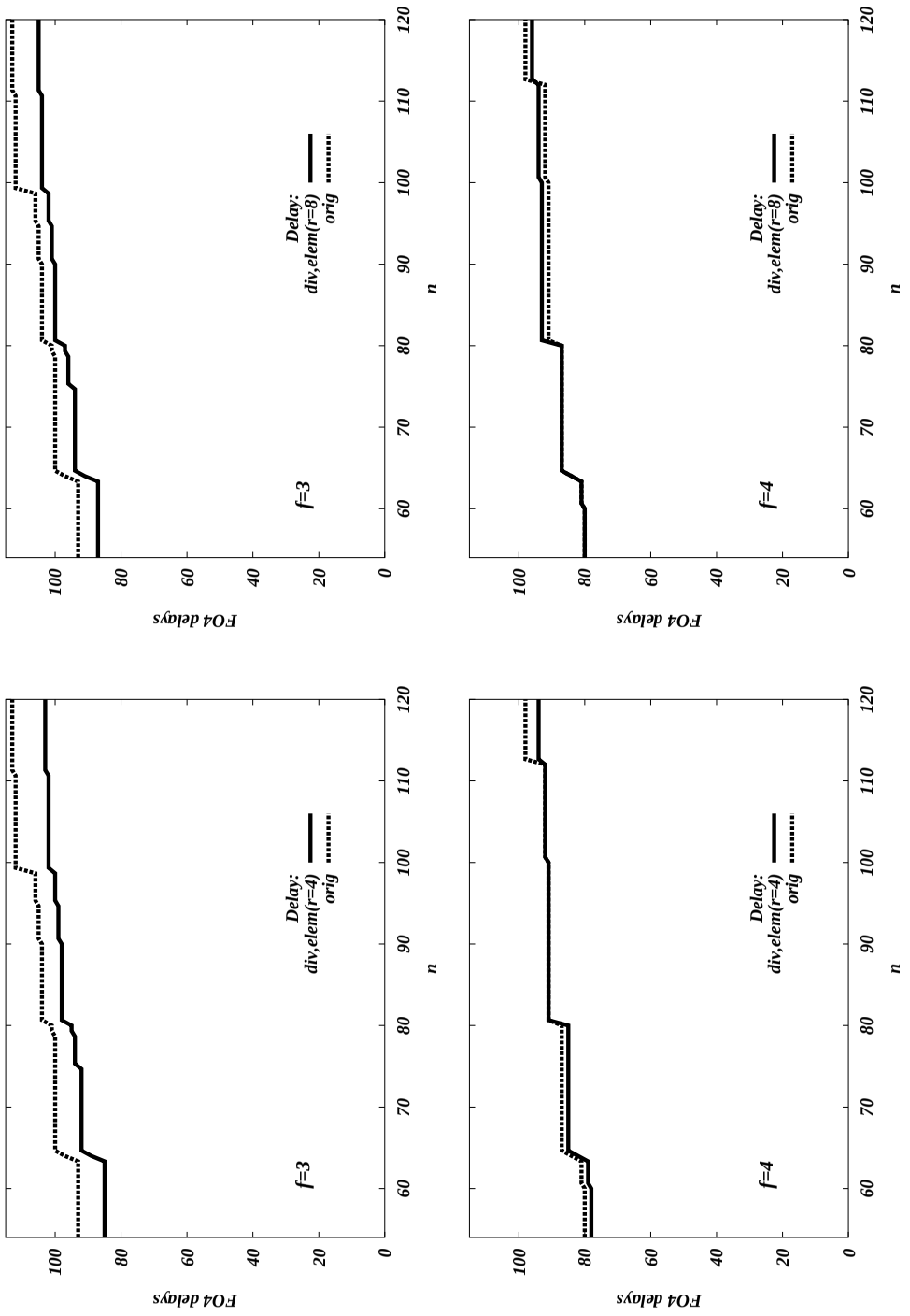


Figure 6.2: Time delay versus significand width for different f and r values.

Chapter 7

Putting it all together

As the analysis in the previous chapters showed, the proposed system is suited for high speed floating point units that are designed for double precision or larger operand width. In the following few sections an attempt is made to quantify the impact of using this proposed system within a processor instead of a conventional FPU. The two axes of comparisons where the impact is evaluated are the speed and the area.

7.1 Speed impact of the overall system

According to the analysis in chapter 4 the time delay of the proposed adder is given by:

$$\begin{aligned}\tau_{add} &= 16 \\ &\quad + 2 \times \lceil \log_{f-1}(\lceil \frac{expWF}{f} \rceil - 1) \rceil \\ &\quad + 2 \times \lceil \log_4(\lceil \frac{n}{r} \rceil \times (r+1) - 1) \rceil \\ &\quad + \lceil \log_2(\lceil \frac{n}{r} \rceil) \rceil \\ &\quad + 2 \times \lceil \log_{f-1}(\lceil \frac{r+1}{f} \rceil - 1) \rceil + \lceil \log_4(r+1) \rceil\end{aligned}$$

while the best conventional design for large significand sizes has a delay of:

$$\begin{aligned}\tau_{RL} &= 15 + \lceil \log_4(n) \rceil + \lceil \log_2(n) \rceil \\ &\quad + 2 \times \lceil \log_{f-1}(\lceil \frac{expW}{f} \rceil - 1) \rceil + 2 \times \lceil \log_{f-1}(\lceil \frac{n}{f} \rceil - 1) \rceil\end{aligned}$$

Let us define the relative speed improvement of the adder as $s_a = (\tau_{RL} - \tau_{add})/\tau_{RL}$. This is a

function of n, f and r. For the purpose of this speedup study n is varied between 53 and 120 since this is the region of applicability of the proposed system. Two possible values for f are considered, namely, $f = 3$ and $f = 4$ since those are the practical values in static CMOS technologies. Finally r is set to either 4 or 8.

Similarly a relative speed improvement for the multiplier s_m can be defined. The time delay of the proposed design is:

$$\begin{aligned}\tau_{mul} &= 12 \\ &\quad + \lceil \log_4(\lceil \frac{n}{r} \rceil \times (r+2)) \rceil \\ &\quad + 3\lceil \log_2(\lceil \frac{n}{2} \rceil + 3) \rceil \\ &\quad + 2 \times \lceil \log_{f-1}(\lceil \frac{r+1}{f} \rceil - 1) \rceil \\ &\quad + \lceil \log_4(\lceil \frac{n}{r} \rceil \times (r+1) - 1) \rceil\end{aligned}$$

while the time delay of the conventional designs is given by:

$$\begin{aligned}\tau_{con} &= 6 + \lceil \log_4(n) \rceil + 3(\lceil \log_2 \lceil \frac{n}{2} \rceil \rceil) \\ &\quad + 2 \times \lceil \log_{f-1}(\lceil \frac{n}{f} \rceil - 1) \rceil + \lceil \log_4(n) \rceil\end{aligned}$$

The definition s_m is then $s_m = (\tau_{con} - \tau_{mul})/\tau_{con}$.

As discussed in chapter 6, the division and elementary functions unit has practically the same time delay as the conventional design and hence its speed improvement is considered to be zero. We also consider that the speed improvement of the other instructions (load, store,...) is zero. Hence, it is only the adder and multiplier units that contribute to the overall improvement.

In chapter 1 two different studies of instruction mix were presented with different results. One has the add/subtract unit accounting for 40% of the floating point instructions while the share of the multiply is 37%. The other has the add/subtract at 24% and the multiply at 19%. If the percentage of the add/subtract is p_a and that of the multiplier is p_m then the overall speed improvement of the system can be estimated as:

$$\begin{aligned}s_{sys} &= p_a s_a + p_m s_m \\ &= p_a \frac{\tau_{RL} - \tau_{add}}{\tau_{RL}} + p_m \frac{\tau_{con} - \tau_{mul}}{\tau_{con}}\end{aligned}$$

Fig. 7.1 shows a comparison between the different cases of f and r when n varies from 53 to 120. The percentage of the adder and multiplier are taken as given by the two studies mentioned in chapter 1. Since the second study gives percentages for the adder and multiplier that are almost half of those given in the first study, it is not surprising that the curve corresponding to the second

study is almost at half the value of the curve corresponding to the first study. Hence there is nothing to learn by comparing the curves corresponding to the two studies. What seems interesting is to compare the curves of the studies to the relative speed improvements of the individual adder and multiplier.

Around $n = 60$, the proposed multiplier performs slightly worse than the conventional multipliers which causes a 'negative' improvement. However, in the cases of $f = 3$, this drop in the speed of the multiplier is more than offset by the improvement of the adder due to the higher percentage of use of the adder and its higher speed improvement. In the cases of $f = 4$, the adder itself does not have an improvement around $n = 60$.

As n increases the improvement increases but it is clear that significant improvements are present only in the case of $f = 3$, i.e. when the technology forces the designer to use fan-in limited gates. The value of r should be kept to 4 if a large improvement is required.

As the percentage of use of the adder and multiplier change the conclusions drawn above might change. For example, the limiting case of very high use of the adder will practically correspond to the curve of s_a. That s_a curve shows a improvement of more than 5% at $n = 120$ even in the worst case of $f = 4$ and $r = 8$. It can reach an improvement of more than 16% in the favorable case of $f = 3$ and $r = 4$.

The time delay equations developed through this work and the relative speed improvement relations explained in this chapter are tools enabling the designer of a floating point unit to evaluate the different choices. These tools help us to identify the region of applicability of the proposed system depending on the instruction mix assumptions.

7.2 Area impact of the overall system

It is not possible to accurately predict the area of the proposed designs without performing a physical layout of the transistors. This is the third level of evaluation according to the classification given in chapter 3 and it is beyond the scope of the current work.

It is still however possible to estimate some of the impact of the proposed system. The register file of the FPU is clearly larger and the additional area needed is approximately proportional to the increase in the width of the floating point numbers. That increase of the width also translates into a wider data-path for the functional units. the increase of the area of the functional units is not necessarily proportional to that increased width though. For example, in the proposed design of the adder, the shifters are simpler than in the conventional adders. Another example is the logic for carry computation in the final carry propagate adder in both the floating point adder and multiplier are eliminated in the proposed designs. Despite the increased data-path width these hardware simplifications might make the total area of the functional units comparable to conventional ones. While unable to confirm this now we can quantify the relative increase in the width.

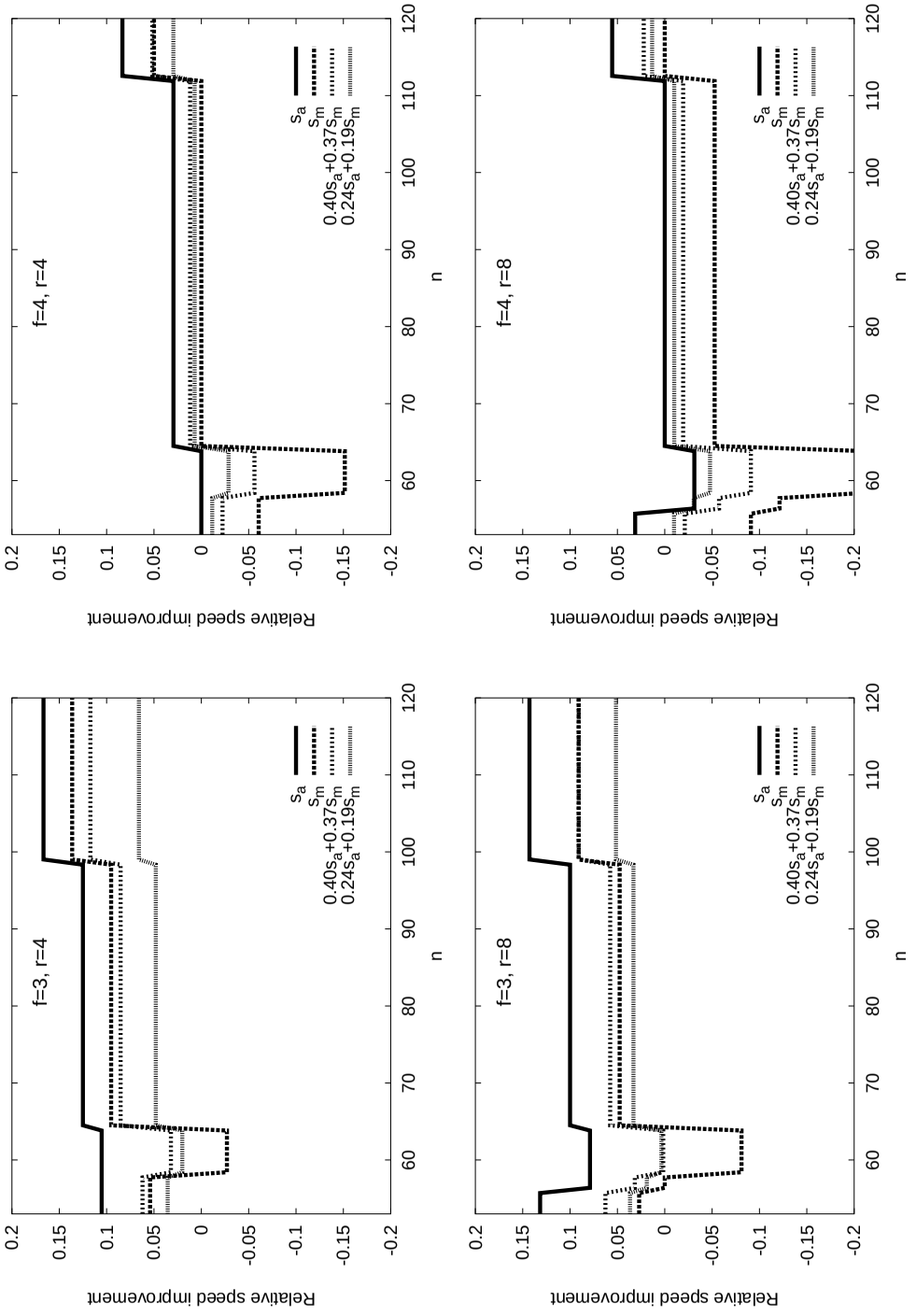


Figure 7.1: Relative speed improvement versus significand width.

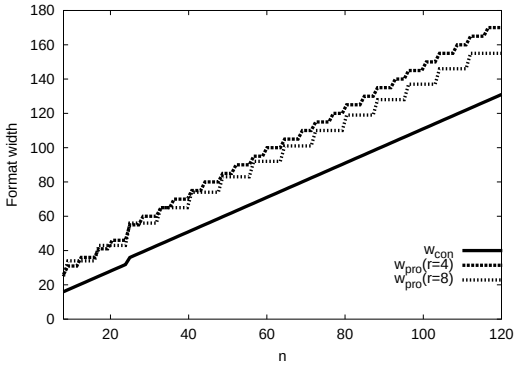


Figure 7.2: Width of the floating point number versus the significand width.

As mentioned in section 4.3, if the significand width of an floating point number inspired by the IEEE formats (including the hidden 1) is n then the width of the significand in the proposed format is equal to $\lceil \frac{n}{r} \rceil \times (r+1) - 1$. In addition to that 5 bits are needed for the guard, round and sticky digits. As for the exponent width of the proposed format $expWF$, it is wider than the exponent width of the IEEE format $expW$. Hence, for $n \leq 24$ $expW$ is taken as 8 and $expWF$ is taken as 11. Otherwise, $expW$ is 11 and $expWF$ is 15.

The full width of the number of an IEEE-like format is then

$$w_{con} = 1 + (expW) + (n-1) = expW + n$$

The first 1 being for the sign bit and the deducted 1 is because of the hidden one. In the case of the proposed format there is no hidden one and the full width is

$$\begin{aligned} w_{pro} &= 1 + expWF + \lceil \frac{n}{r} \rceil \times (r+1) - 1 + 5 \\ &= 5 + expWF + \lceil \frac{n}{r} \rceil \times (r+1) \end{aligned}$$

Fig. 7.2 shows the change of the format's width over the range of $n = 8$ to $n = 120$ with the cases of $r = 4$ and $r = 8$. There is a jump in the size at $n = 24$ due to the sudden change of the exponent besides this, the width of the conventional floating point numbers is a smooth line. On the other hand, the width of the proposed format increases in steps because it increases by full digits of $r + 1$ bits each time. Hence the steps in the case of $r = 8$ are wider and higher than in the case of $r = 4$.

We can define the relative increase in width as $w_{rel} = (w_{pro} - w_{con})/w_{con}$. Fig. 7.3 presents this relative increase in the width. As expected the use of redundancy entails a very high cost at small significand width. On the other hand, in the region of applicability of the proposed format (double precision and higher) the relative increase in the width is limited to about 40% or less.

To re-iterate the relative increase in width is directly proportional to the increase in area of the

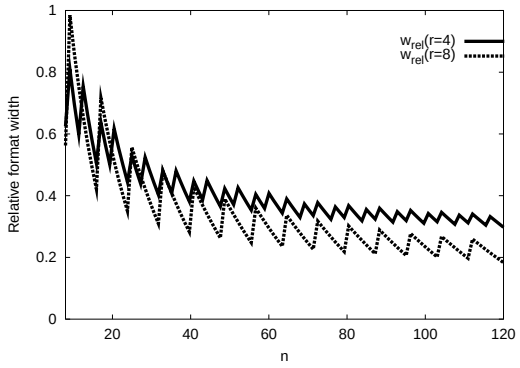


Figure 7.3: Relative increase in the width of the proposed format.

Table 7.1: Time delay and width increase at $n = 80$ and $f = 3$.

		$r = 4$	$r = 8$
Improvement	Addition	12.5%	10%
	Multiplication	9.5%	4.8%
	Div. and elem. functions	5.94%	3.96%
Width increase		31.87%	20.88%

register file but not necessarily to the increase in area of the functional units. This relative increase in the width could be taken as an upper bound on the increase in area.

7.3 Conclusions from a system perspective

The proposed system provides designers of floating point units with more options that can enable them to trade-off an amount of extra area for an improved speed. So it is possible for example to achieve an improvement of 5 to 10% in speed while sacrificing at most 20 to 30% increase in area. To give a concrete example, Table 7.1 shows the improvements in time delay and the potential increase in width for the case of $n = 80$ and $f = 3$. Whether the trade-off is desirable or not is a question of cost and benefit analysis specific to the design required. This chapter presented tools to help the designer make such a decision.

Chapter 8

Conclusions and future work

The work presented in this research aimed at understanding the issues concerning the design of a floating point system using redundant digits. Through the analysis, the region of the design space where such a system outperforms conventional systems was identified. An attempt was made to get as close as possible to the real performance by implementing both the adder and the multiplier at the transistor level. This proved to be a tedious and time consuming task but very beneficial at the end. This task enabled the verification of the correctness of the algorithm as well as the determination of the approximate speed of operation. The prediction by the analytical model and the simulation result show that the proposed system has an advantage over the conventional systems at large significand sizes. The cost for this speed improvement is predicted to be some increase in the area of the floating point unit. The amount of this increase cannot be exactly predicted except with a full layout of the circuits.

In short, the contributions of this work include:

1. The proposed semi-redundant format is not a fully redundant format where the size of the significand doubles but it retains a useful redundancy that improves the speed of the circuits. The use of a hexadecimal exponent allows for a further speed improvement by simplifying the shifters.

2. The postponed rounding technique allows for faster circuits where the rounding delay is hidden by the exponent difference calculation.

3. A new technique for the leading digit detection is derived to suit the proposed redundant format. Its underlying ideas of doing the coarse detection only and leaving the fine adjustment to the rounding stage are applicable to other redundant formats.

4. A new Booth recoding scheme enables "negative" valued bits as inputs and can directly handle the redundant format.

5. A parametric time delay model that is quite powerful in predicting the time delay of large floating point units yet its use is simple. Such a model is useful for comparisons and its estimations are not very far off from more accurate (but much slower and not parametric) simulation results.

These ideas barely opened the door for a host of other interesting points for further research.

- Some machines implement both decimal and binary number systems. A simple modification in the logic for carry generation in the SD adders enables both number systems to use the same hardware unit with minimal extra area. In the case of decimal, each four bits will represent a digit in $\{-9, -8, \cdots, 8, 9\}$ and the assumed radix will be 10 instead of 16.

- Adders and multipliers for various applications (specially multimedia applications) are partitioned to allow for the same hardware to do any of a 64×64 operation or two simultaneous 32×32 operations or even four simultaneous 16×16 operations. With the redundant format already partitioned into digits, it is easy to segment the adder into two or four parts. We just add at the required breaking points some multiplexers to either stop or pass the carries; hence effectively breaking the large unit into smaller ones. Segmenting the multiplier or the divider is not as simple as that but still it can be done with probably less additional hardware than in the conventional designs.

- If a layout is done for the complete floating point unit (including division and elementary functions) then accurate speed, area and power estimates are possible. A fabricated and tested chip is the litmus test for redundant digit floating point designs.

- With more details being put in the proposed design at the layout level there are some questions that should be checked at the algorithmic level to probably introduce further enhancements:

  - Is it possible to use a leading digit *prediction* scheme in the adder?
  - Can we reduce the hardware of the cancellation path instead of requiring $A - B$, $B - A$, $A - shift(B)$ and $B - shift(A)$?
  - Is there a benefit in using the idea of switching E and P instead of generating mX in the multiplier?
  - Can all the algorithms used be formally proved as correct?

- On the more innovative side, maybe a comparisons with other types of redundancies (say where the extra bits are not equidistant) or other number representations (say continued fractions) will prove beneficial to further improve the performance of the floating point unit.

- In fact, in the future, we should think of performance as not just equivalent to speed but as a general function including speed, area, power, reliability, testability, serviceability, ... and

redundancy can help in a few of those areas indeed. Redundant representations have been used for error detection and correction. A scheme can be designed where redundancy serves for both speeding up the computation and making it less error prone. Redundancy also shortens the carry propagation path, does this mean a smaller number of the circuit nodes switching and a smaller power consumption? Can redundant representation produce high speed systems consuming less power? These are open questions.

As one of my teachers once told me, "When you start researching a topic, it is like an avalanche afterwards!" This is true, once you start something it opens the door for you to do many more things. The amount of knowledge that we, humanity, attained is very limited and we can all, and in fact we must all, try to learn more.

«وَمَا أُوتِيتُم مِنَ ٱلْعِلْمِ إِلَّا قَلِيلًا»، خِتَامُ ٱلْآيَةِ ٨٥ مِنْ سُورَةِ ٱلْإِسْرَاءِ

"And you did not get from knowledge except a little" translation from The Holy Quran, the last part of verse 85 of Surat-Al-IsrA' (the night journey, 17)

Appendix A

Leading digit detection

The challenge in the leading digit detection part of the proposed design is due to the different patterns that result in leading insignificant digits. The partial compression that achieves the coarse adjustment by detecting the patterns

$$\begin{array}{rrrcrcccccccccccc} 1 & -15 & -15 & \cdots & -15 & l & m & \cdots & = & 0 & 0 & 0 & \cdots & 1 & l & m & \cdots \\ -1 & 15 & 15 & \cdots & 15 & l & m & \cdots & = & 0 & 0 & 0 & \cdots & -1 & l & m & \cdots \end{array}$$

depends on two mechanisms: N-recoding and P-recoding.

As mentioned in section 4.2.1, for two consecutive digits of the result,

$$\begin{array}{ccccccccc} \cdots & s_{i_3} & s_{i_2} & s_{i_1} & s_{i_0} & & s_{i-1_3} & s_{i-1_2} & \cdots \\ & s_{i_4} & & & & s_{i-1_4} & & & \end{array}$$

the N-recoding is defined as reseting s_{i-1_4} and s_{i_0} to 0 if they were both 1. Hence the output bits are $s^n_{i_0} = s_{i_0}\overline{s}_{i-1_4}$ and $s^n_{i-1_4} = \overline{s}_{i_0}s_{i-1_4}$ while the remaining bits of the digit pass unchanged.

The P-recoding eliminates the case of insignificant leading -1 followed by positive digits. Referring to the two consecutive digits above, if $s_{i_0} = 1$ and $s_{i-1_4} = 0$ then we can think of s_{i-1_4} as equal to two parts $+1$ and -1. We separate the $+1$ and add it to the higher order digit s_i while the -1 is kept with the lower order digit s_{i-1} as its new s_{i-1_4}. This separation occurs only if the whole digit s_{i-1} is not exactly equal to zero. Otherwise, the new s_{i-1} becomes -16 which is undesirable as will be explained shortly. Applying P-recoding to the case of repeated digits of 15 the result is:

digits	0	-1	15	$\cdots$	15	l	$\cdots$
equiv.	0	1111	1111	$\cdots$	1111	$l_3l_2l_1l_0$	$\cdots$
bits	1	0	0	$\cdots$	l_4	$\cdots$	$\cdots$
result	0	0	0	$\cdots$	-1	l	$\cdots$

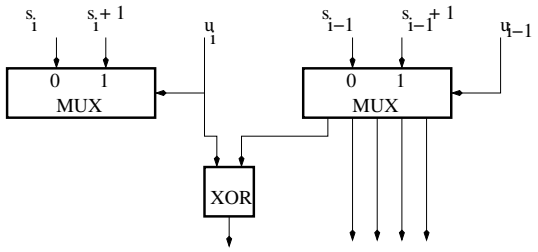


Figure A.1: P-recoding implementation.

In this, the digit l is assumed negative ($l_4 = 1$) and the whole result is negative at the end. If l is positive then one more digit becomes zero and the result is $0\,(l-16)\cdots$. Note that even in this case, the sign of the result is still negative since $\mid l \mid \leq 15$. In general due to the choice of base and possible values in this number system, any number has the sign of its leading non-zero digit [31]. The N and P-recodings do not alter that.

The condition mentioned above for the P-recoding to change the bits is $s_{i_0} = 1$, $s_{i-1_4} = 0$ and $\overline{z}_{i-1} = 1$, where $\overline{z}_{i-1}$ is an indicator to show if the digit s_{i-1} is not zero. Let $u_i = s_{i_0}\overline{s}_{i-1_4}\overline{z}_{i-1}$, then if $u_i = 1$ the output of the P-recoding for digit i is $s_i + 1$ instead of s_i. Obviously, u_i could be added to s_i or, better, it could be used as a select line in a multiplexer which has s_i and $s_i + 1$ as its inputs. This u_i signal also affects the most significant bit of the lower adjacent digit s_{i-1}. If $u_i = 0$ then the output of the P-recoding for this bit, $s^p_{i-1_4}$, is determined just by the outcome of the multiplexer choosing between s_{i-1} and $s_{i-1} + 1$ depending on u_{i-1}. If, on the other hand, $u_i = 1$ then the two possible cases of u_{i-1} need to be analyzed.

$\mathbf{u}_{i-1} = 0$**:** $s^p_{i-1_4} = 1 = \overline{s}_{i-1_4}$ (remember that $s_{i-1_4} = 0$ for $u_i = 1$).

$\mathbf{u}_{i-1} = 1$**:** then two conditions are possible:

sign bit of $s_{i-1} + 1$ **is** 0**:** Then, as above, $s^p_{i-1_4} = 1$.

sign bit of $s_{i-1} + 1$ **is** 1**:** This means that due to the added 1 an overflow occurred which has a value of +1. That positive overflow is canceled out by the −1 resulting from the P-recoding splitting of the original 0 in s_{i-1_4}, thus $s^p_{i-1_4} = 0$.

Hence in all the cases, if $u_i = 1$ the resulting $s^p_{i-1_4}$ bit is the inverse of the bit coming out of the multiplexer choosing between s_{i-1} and $s_{i-1} + 1$. This leads to the possible implementation shown in Fig. A.1.

As is the case for the N-recoding an "out of bound" digit value can occur. In this case, a value of +16 results if the pattern of digits k 15 l with $k_0 = 0$ and $l_4 = 0$ are entered into a P-recoder. Again, this is not problematic since this recoded format is only within the LDD circuits and the position of the leading non-zero digit is correctly detected as described below. However, this is why it was

important to note above that in P-recoding, this split of the zero in s_{i-1_4} occurs only if s_{i-1} is not exactly equal to zero in order to prevent the new s_{i-1} from becoming -16. Otherwise, a difficulty arises: how to distinguish between the out of bound $+16$ and the -16 resulting from subtracting 16 from a digit that is already zero?

Either the N-recoding or the P-recoding can be done first, there is no strict order:

$P(N(s))$ The N-recoding detects the case of $s_{i_0}, s_{i-1_4} = 1, 1$ and can produce a digit of value -16 as an anomaly in the case of $\cdots(even)\ -15\ (-ve)\cdots$. However, this does not affect the following P-recoding and the result of the N-recoding gives the correct position and sign of the leading digit since the N-recoding results in either $s^n_{i_0}, s^n_{i-1_4} = 0, 0$ (normal case) or $0, 1$ (anomaly) which are passed as is in a P-recoding.

$N(P(s))$ The P-recoding detects the case of $s_{i_0}, s_{i-1_4} = 1, 0$ and can produce a digit of value $+16$ as an anomaly in the case of $\cdots(even)\ 15\ (+ve)\cdots$. This does not affect the following N-recoding because the P-recoding results in either $s^p_{i_0}, s^p_{i-1_4} = 0, 1$ (normal or anomaly) or $0, 0$ (case of $s_{i-1} + 1$ overflow) which are passed as is in an N-recoding. However, although the result has the correct position for the leading digit, it might be assumed negative if the first digit is that anomaly of $+16$. This problem is also present in the case of $P(N(s))$ above.

The problem of having the correct position but possibly the wrong sign can be solved by keeping the sign information of each digit in another bit. Let us call it n_i which is 1 if the digit is negative. So, before any recoding, $n_i = s_{i_4}$. The following tree allows us to deduce the expression giving n^p_i the output after the P-recoding.

$$
\left\langle
\begin{array}{lll}
n_i = 0 & \left\langle
\begin{array}{ll}
u_{i+1} = 0 & \left\langle
\begin{array}{ll}
s^p_{i_4} = 0 & n^p_i = 0, \text{for } s_i > 0 \text{ and } s_i + 1 < 16 \\
s^p_{i_4} = 1 & n^p_i = 0, \text{for } s_i = 15 \text{ and } u_i = 1
\end{array}
\right. \\
u_{i+1} = 1 & \left\langle
\begin{array}{ll}
s^p_{i_4} = 0 & n^p_i = 0, \text{for } s_i = 15 \text{ and } u_i = 1 \\
s^p_{i_4} = 1 & n^p_i = 1, \text{for } s_i > 0 \text{ and } s_i + 1 < 16
\end{array}
\right.
\end{array}
\right. \\
n_i = 1 \ \Rightarrow \ u_{i+1} = 0 & \left\langle
\begin{array}{ll}
s^p_{i_4} = 0 & n^p_i = 0, \text{for } s_i = -1 \text{ and } u_i = 1 \\
s^p_{i_4} = 1 & n^p_i = 1, \text{for other cases with } s_i < 0
\end{array}
\right.
\end{array}
\right.
$$

So digit i is positive if it was already positive before the P-recoding and either $u_{i+1} = 0$ or $u_{i+1} = 1$ *and* $s^p_{i_4} = 0$. The condition for it to be negative is:

$$
\begin{aligned}
n^p_i &= n_i s^p_{i_4} + \overline{n}_i u_{i+1} s^p_{i_4} \\
&= (n_i + u_{i+1}) s^p_{i_4}
\end{aligned}
$$

In the N-recoding, the equation giving n_i^n is simpler to deduce. Basically, if $s_{i+1_0}s_{i_4} = 1$ then $n_i^n = 0$ otherwise $n_i^n = n_i$. This gives:

$$n_i^n = n_i\overline{s_{i+1_0}s_{i_4}}$$

If, for each digit after the recodings, another indicator $\overline{z}_i$ is kept to indicate if the digit is not zero then the final decision of the LDD is based on those n_i's and $\overline{z}_i$'s. This indicator is given by $\overline{z}_i = \overline{s_{i_4} + s_{i_3} + s_{i_2} + s_{i_1} + s_{i_0}}$. The first digit that has $\overline{z}_i = 1$ is the leading digit and its sign is the corresponding n_i. The bit s_{i_4} is included in $\overline{z}_i$ because of the ± 16 anomalies.

For the special case of $000(1)(-ve)\cdots$, the LDD indicates a shift up to the digit after the 1 and indicates that its sign is positive ($n = 0$ after the N-recoding). The left shifter shifts to the correct position and uses the information about the sign given by the LDD to set the sign bit of the resulting most significant digit accordingly. The same goes for the case of $000(-1)(+ve)\cdots$. The reason these two special cases are handled in this manner is that the rounding stage is supposed to work only on the fine adjustment with at most one bit location shift. As an example, if the result is $000(1)(-14)\cdots$ and the LDD did not function as described above but rather indicated that the leading digit is the 1 then in the rounding stage the "fine" adjustment evaluates:

$$\begin{array}{cccccccccc} 0001 & 0010 & xxxx & \cdots & \Rightarrow & 0000 & 0010 & xxxx & \cdots \\ 1 & x & x & & & 0 & x & x & \end{array}$$

which is a shift by three bit positions. The rounding stage is kept simpler by making the maximum shift there equal to one bit location. This condition to have at most one bit location shift is insured in the cancellation path by the functionality of the LDD as described above. In the far path this condition is insured by the design of the adder used there.

If the end result of the cancellation path is negative then it should be inverted and the sign of the whole floating point number changed. Otherwise, to improve the speed, the SD adder can be designed to output the sum digits and their negatives. Both of those are then left shifted in parallel to the position determined by the LDD using two shifters. One of them is finally chosen depending on the sign of the end result. The implementation of the floating point adder presented in this work uses this latter approach.

The advantage of having the P-recoding before the N-recoding is that the $s_i + 1$ required for it can be calculated in the adder in parallel. This saves some time compared to doing it sequentially on the outcome of the N-recoding. However, the disadvantage is that the adder's output can be either $x_i + y_i$, $x_i + y_i + 1$ or $x_i + y_i - 1$ depending on the carry into this digit position. Hence, the needed $s_i + 1$ could be either $x_i + y_i$, $x_i + y_i + 1$ or $x_i + y_i + 2$ which means that we need to have four addition circuits to calculate the different possibilities and use three of those for each of the outcomes s_i and $s_i + 1$. Other ways may be devised to eliminate the need for the fourth addition

circuit as described below. Some designers may opt to have the faster implementation regardless of its complexity and others may choose the simpler one. That is why both, the case of implementing $P(N(s_i))$ and that of $N(P(s_i))$ are outlined here with the case of $P(N(s_i))$ presented first because it might be conceptually easier.

N-recoding first

In this case the outcome of the N-recoding is:

$$\begin{aligned}
s^n_{i-1_4} &= \overline{s}_{i_0} s_{i-1_4} \\
s^n_{i_0} &= s_{i_0} \overline{s}_{i-1_4} \\
s^n_{i_1} &= s_{i_1} \\
s^n_{i_2} &= s_{i_2} \\
s^n_{i_3} &= s_{i_3} \\
n^n_i &= \overline{s}_{i+1_0} s_{i_4} \\
\overline{z}^n_i &= \overline{s}_{i+1_0} s_{i_4} + s_{i_3} + s_{i_2} + s_{i_1} + s_{i_0} \overline{s}_{i-1_4}
\end{aligned}$$

For the following P-recoding, the condition u_i to use $s^n_i + 1$ becomes then

$$\begin{aligned}
u^{pn}_i &= s^n_{i_0} \overline{s^n}_{i-1_4} \overline{z}^n_{i-1} \\
&= (s_{i_0} \overline{s}_{i-1_4})(\overline{\overline{s}_{i_0} s_{i-1_4}})(\overline{s}_{i_0} s_{i-1_4} + s_{i-1_3} + s_{i-1_2} + s_{i-1_1} + s_{i-1_0} \overline{s}_{i-2_4}) \\
&= s_{i_0} \overline{s}_{i-1_4} (s_{i-1_3} + s_{i-1_2} + s_{i-1_1} + s_{i-1_0} \overline{s}_{i-2_4})
\end{aligned}$$

If $u^{pn}_i = 1$ then one is added to s^n_i and the most significant bit of the result of the similar addition for s^n_{i-1} is inverted. Hence, $s^{pn}_i = s^n_i + u_i - (16)u_{i+1}$. The final sign of each digit is given by $n^{pn} = (n^n_i + u^{pn}_{i+1}) s^{pn}_{i_4}$. For this, as well as for the decision of whether the digit after the recoding is zero or not, the bits of s^{pn}_i are needed. These are basically the outcome of adding a single bit u^{pn}_i to s^n_i and a special treatment for $s^{pn}_{i_4}$. Either an adder could be used for this, or an evaluation of the boolean expression giving each of the bits of s^{pn}_i could be made. The resulting equations are:

$$\begin{aligned}
s^{pn}_{i_0} &= s^n_{i_0} \oplus u^{pn}_i \\
s^{pn}_{i_1} &= s^n_{i_1} \oplus s^n_{i_0} u^{pn}_i = s^n_{i_1} \oplus u^{pn}_i \\
s^{pn}_{i_2} &= s^n_{i_2} \oplus s^n_{i_1} u^{pn}_i \\
s^{pn}_{i_3} &= s^n_{i_3} \oplus s^n_{i_2} s^n_{i_1} u^{pn}_i
\end{aligned}$$

$$s_{i_4}^{pn} = s_{i_4}^{n} \oplus s_{i_3}^{n} s_{i_2}^{n} s_{i_1}^{n} u_i^{pn} \oplus u_{i+1}^{pn}$$

By using the identities $a \oplus (ab) = a\overline{b}$, $a(a \oplus b) = a\overline{b}$, $(a \oplus b) + (c \oplus ab) + (d \oplus abc) = \overline{abcd}(a+b+c+d)$ and substituting in terms of s_i then simplifying, the results become:

$$\begin{aligned}
u_i^{pn} &= s_{i_0}\overline{s}_{i-1_4}(s_{i-1_3} + s_{i-1_2} + s_{i-1_1} + s_{i-1_0}\overline{s}_{i-2_4}) \\
s_{i_4}^{pn} &= (\overline{s}_{i+1_0} s_{i_4} + u_{i+1}^{pn}) \oplus s_{i_3} s_{i_2} s_{i_1} u_i^{pn} \\
n_i^{pn} &= (\overline{s}_{i+1_0} s_{i_4} + u_{i+1}^{pn})(\overline{s_{i_3} s_{i_2} s_{i_1} u_i^{pn}}) \\
\overline{z}_i^{pn} &= s_{i_0}\overline{s}_{i-1_4}(\overline{s_{i-1_3} + s_{i-1_2} + s_{i-1_1} + s_{i-1_0}\overline{s}_{i-2_4}}) \\
&\quad + \overline{s_{i_3} s_{i_2} s_{i_1} u_i^{pn}}(s_{i_3} + s_{i_2} + s_{i_1} + u_i^{pn}) \\
&\quad + (\overline{s}_{i+1_0} s_{i_4} + u_{i+1}^{pn})(\overline{s_{i_3} s_{i_2} s_{i_1} u_i^{pn}}) \\
&\quad + (\overline{\overline{s}_{i+1_0} s_{i_4} + u_{i+1}^{pn}}) s_{i_3} s_{i_2} s_{i_1} u_i^{pn}
\end{aligned}$$

The equations for $s_{i_4}^{pn}$ and n_i^{pn} fit the expected behavior. For $s_{i_4}^{pn}$, it is 1 if the original digit is negative and, passing through the N-recoding, it is not affected because the higher digit is even (the $\overline{s}_{i+1_0} s_{i_4}$ part) or if due to the P-recoding a 1 is inserted there (the u_{i+1}^{pn} part). These two parts are mutually exclusive and, in fact, the 1 that they produce has a negative value. To either of these a positive one can be added if $u_i^{pn} = 1$ and there is a path for it to propagate over the bits of the digit (condition for that is $s_{i_3} s_{i_2} s_{i_1} s_{i_0}$). Similarly, for n_i^{pn}, the digit is negative ($n_i^{pn} = 1$) if either of the first two parts discussed above introducing a negative 1 into $s_{i_4}^{pn}$ is true and the condition for the positive 1 propagation is false. This insures that n_i^{pn} remain 0 when $s_{i_4}^{pn}$ is 1 only due to the propagating 1 since in that case $s_{i_4}^{pn}$ is not of negative value anymore and the whole digit is positive.

P-recoding first

Depending on the carry into the adder digit position being considered, the output could either be $x_i + y_i$, $x_i + y_i + 1$ or $x_i + y_i - 1$. Hence, if the P-recoding is done first then, within the adder, some circuits to calculate $x_i + y_i + 2$ are needed. Then, the carry could be used to determine the value corresponding to $s_i + 1$. The condition for u_i in this case becomes $u_i^p = s_{i_0}\overline{s}_{i-1_4}\overline{z}_{i-1} = s_{i_0}\overline{s}_{i-1_4}(s_{i-1_3} + s_{i-1_2} + s_{i-1_1} + s_{i-1_0})$. This u_i^p could then be used to select between s_i and $s_i + 1$. However, instead of adding extra circuitry and selecting later, the same idea of calculating s_i^{pn} in terms of the bits of s_i developed for the case of "N-recoding first" can be applied. This produces:

$$\begin{aligned}
u_i^p &= s_{i_0}\overline{s}_{i-1_4}(s_{i-1_3} + s_{i-1_2} + s_{i-1_1} + s_{i-1_0}) \\
s_{i_0}^p &= s_{i_0} \oplus u_i^p = s_{i_0}\overline{u_i^p}
\end{aligned}$$

$$
\begin{aligned}
s^p_{i_1} &= s_{i_1} \oplus u^p_i \\
s^p_{i_2} &= s_{i_2} \oplus s_{i_1} u^p_i \\
s^p_{i_3} &= s_{i_3} \oplus s_{i_2} s_{i_1} u^p_i \\
s^p_{i_4} &= s_{i_4} \oplus s_{i_3} s_{i_2} s_{i_1} u^p_i \oplus u^p_{i+1} \\
&= (s_{i_4} + u^p_{i+1}) \oplus s_{i_3} s_{i_2} s_{i_1} u^p_i \\
n^p_i &= (s_{i_4} + u^p_{i+1}) s^p_{i_4} \\
&= (s_{i_4} + u^p_{i+1})(\overline{s_{i_3} s_{i_2} s_{i_1} u^p_i})
\end{aligned}
$$

Then the following N-recoding produces

$$
\begin{aligned}
s^{np}_{i_0} &= s^p_{i_0} \overline{s^p}_{i-1_4} \\
&= s_{i_0} \overline{u^p_i} (\overline{(s_{i-1_4} + u^p_i) \oplus s_{i-1_3} s_{i-1_2} s_{i-1_1} u^p_{i-1}}) \\
&= s_{i_0} \overline{u^p_i} (s_{i-1_4} \oplus (\overline{s_{i-1_3} s_{i-1_2} s_{i-1_1} u^p_{i-1}})) \\
s^{np}_{i_1} &= s^p_{i_1} \\
s^{np}_{i_2} &= s^p_{i_2} \\
s^{np}_{i_3} &= s^p_{i_3} \\
s^{np}_{i_4} &= \overline{s^p}_{i+1_0} s^p_{i_4} \\
&= (\overline{s}_{i+1_0} + u^p_{i+1})((s_{i_4} + u^p_{i+1}) \oplus s_{i_3} s_{i_2} s_{i_1} u^p_i) \\
&= \overline{s}_{i+1_0} (s_{i_4} \oplus s_{i_3} s_{i_2} s_{i_1} u^p_i) + u^p_{i+1} (\overline{s_{i_3} s_{i_2} s_{i_1} u^p_i}) \\
n^{np}_i &= n^p_i (\overline{s^p_{i+1_0} s^p_{i_4}}) \\
&= (s_{i_4} + u^p_{i+1}) s^p_{i_4} (\overline{s^p}_{i+1_0} + \overline{s^p}_{i_4}) \\
&= (s_{i_4} + u^p_{i+1})((s_{i_4} + u^p_{i+1}) \oplus s_{i_3} s_{i_2} s_{i_1} u^p_i)(\overline{s_{i+1_0} \overline{u^p}_{i+1}}) \\
&= (\overline{s}_{i+1_0} s_{i_4} + u^p_{i+1})(\overline{s_{i_3} s_{i_2} s_{i_1} u^p_i})
\end{aligned}
$$

Once again, the equations for $s^{np}_{i_4}$ and n^{np}_i fit the expected behavior. However, the u^p_i used here is slightly simpler to evaluate compared to the u^{pn}_i used in the case of "N-recoding first." Finally, the bit indicating if the digit is zero or not is given by

$$
\begin{aligned}
\overline{z}^{np}_i &= s_{i_0} \overline{u^p_i} (s_{i-1_4} \oplus (\overline{s_{i-1_3} s_{i-1_2} s_{i-1_1} u^p_{i-1}})) \\
&\quad + \overline{s_{i_3} s_{i_2} s_{i_1} u^p_i} (s_{i_3} + s_{i_2} + s_{i_1} + u^p_i) \\
&\quad + (\overline{s}_{i+1_0} (s_{i_4} \oplus s_{i_3} s_{i_2} s_{i_1} u^p_i) + u^p_{i+1} (\overline{s_{i_3} s_{i_2} s_{i_1} u^p_i}))
\end{aligned}
$$

The condition $u_i = s_{i_0}\overline{s}_{i-1_4}\overline{z}_{i-1}$ is not a tight condition. The P-recoding causing insignificant digit deletion occurs when $s_{i-1} = 15 = 01111_{bin}$ and $s_i = -1 = 11111_{bin}$ or $s_i = 15$ (assuming that it is preceded by some $s = -1$). So, the strictest condition is $u_i = s_{i_3}s_{i_2}s_{i_1}s_{i_0}\overline{s}_{i-1_4}s_{i-1_3}s_{i-1_2}s_{i-1_1}s_{i-1_0}$. Other conditions between those two extremes can be used, for example $u_i = s_{i_3}s_{i_2}s_{i_1}s_{i_0}\overline{s}_{i-1_4}s_{i-1_3}$. It is interesting to find out the best choice of u_i to minimize the hardware and time delay of the LDD.

To show the possible minimizations, when using $u_i = s_{i_3}s_{i_2}s_{i_1}s_{i_0}\overline{s}_{i-1_4}s_{i-1_3}$ the case of $P(N(s))$ gives:

$$\begin{aligned} u_i^{pn} &= s_{i_3}s_{i_2}s_{i_1}s_{i_0}^n\overline{s}_{i-1_4}^n s_{i-1_3} \\ u_i^{pn} &= s_{i_3}s_{i_2}s_{i_1}(s_{i_0}\overline{s}_{i-1_4})(s_{i_0}+\overline{s}_{i-1_4})s_{i-1_3} \\ u_i^{pn} &= u_i \end{aligned}$$

Hence,

$$\begin{aligned} s_{i_0}^{pn} &= s_{i_0}^n \oplus u_i^{pn} = (s_{i_0}\overline{s}_{i-1_4}) \oplus u_i \\ s_{i_1}^{pn} &= s_{i_1}^n \oplus s_{i_0}^n u_i^{pn} = s_{i_1} \oplus u_i \\ s_{i_2}^{pn} &= s_{i_2}^n \oplus s_{i_1}^n u_i^{pn} = s_{i_2} \oplus u_i \\ s_{i_3}^{pn} &= s_{i_3}^n \oplus s_{i_2}^n s_{i_1}^n u_i^{pn} = s_{i_3} \oplus u_i \\ s_{i_4}^{pn} &= s_{i_4}^n \oplus s_{i_3}^n s_{i_2}^n s_{i_1}^n u_i^{pn} \oplus u_{i+1}^{pn} \\ &= \overline{s}_{i+1_0}s_{i_4} \oplus u_i \oplus u_{i+1} \\ &= (\overline{s}_{i+1_0}s_{i_4} + u_{i+1}) \oplus u_i \end{aligned}$$

which finally yields:

$$\begin{aligned} n_i^{pn} &= (\overline{s}_{i+1_0}s_{i_4} + u_{i+1})\overline{u}_i \\ \overline{z}_i^{pn} &= (\overline{s}_{i+1_0}s_{i_4} + s_{i_3} + s_{i_2} + s_{i_1} + s_{i_0}\overline{s}_{i-1_4})\overline{u}_i \\ &\quad + u_i\overline{u}_{i+1}(s_{i+1_0} + \overline{s}_{i_4}) \end{aligned}$$

It is clear that with this choice of u_i the equations are simpler. The simplifications occurring in $s_{i_{3,2,1,0}}^{pn}$ are due to the fact that with this choice of the condition, if $u_i = 1$ then bits 0 to 3 of s_i are all ones. These bits become all zeros in $s_i + 1$ and $\overline{z}_i^{pn}$ depends only on bit 4 in that case. On the other hand, if $u_i = 0$ then all the bits of s_i^{pn} enter into the determination of $\overline{z}_i^{pn}$.

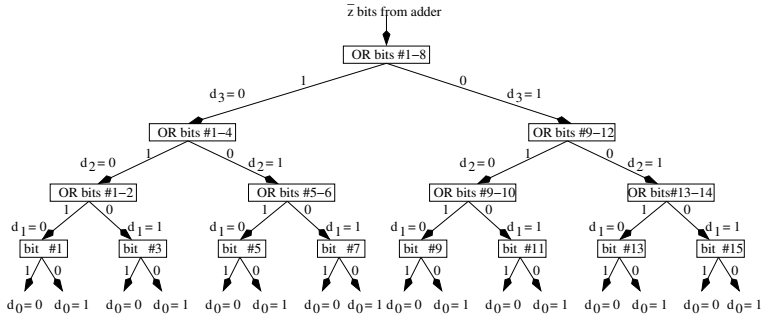


Figure A.2: Encoding the position of the leading digit with a hierarchical tree approach.

Similarly, the case of $N(P(s))$ yields:

$$\begin{aligned} u_i^p &= u_i \\ n_i^{np} &= (\overline{s}_{i+1_0} s_{i_4} + u_{i+1})\overline{u}_i \\ \overline{z}_i^{np} &= (\overline{s}_{i+1_0} s_{i_4} + s_{i_3} + s_{i_2} + s_{i_1} \\ &\quad + s_{i_0}(s_{i-1_4} \oplus \overline{u}_{i-1}))\overline{u}_i + u_i \overline{s}_{i+1_0} \overline{s}_{i_4} \end{aligned}$$

The leading digit detection implemented in this work uses $u_i = s_{i_3} s_{i_2} s_{i_1} s_{i_0} \overline{s}_{i-1_4} s_{i-1_3}$ and $P(N(s))$ to determine the values of n_i and $\overline{z}_i$. The leading non-zero digit is then determined by using the $\overline{z}_i$ bits out of the recodings as inputs to an encoder. That encoder is used to encode the position of the first non-zero digit and this amount is forwarded to the left shifters to normalize the result. Obviously, the n_i bits out of the recodings are shifted as well. The final sign of the number is that of the leading digit as determined by its n bit. Based on this either the result or its negation is chosen and the sign of the whole floating point result is affected.

As the comparative study of leading digit prediction [69] indicates, there are two ways of obtaining an encoded count of the number of leading zeros. Either by a monotonic string of zeros followed by ones or by a hierarchical tree structure. The hierarchical tree [82] is as shown in Fig. A.2. It obtains the encoded position most significant bit first and then allows the shifting operations to be overlapped in time with the decision on the remaining bits. Hence, the result out of the left shifter in Fig. A.3 is available only one multiplexer delay after the last bit of the position is determined.

The LDD used in the floating point adder implemented follows the first approach of having a monotonic string and using a priority encoder to indicate the approximate location of the first non-zero digit. That approach produces an encoding that has only one signal active as shown in Fig. A.4. Such an encoding fits the requirements of the barrel shifters used in the cancellation path. The blocks to the left of Fig A.4 are detailed in Fig. A.5. They produce the n_i and z_i signals given by the $P(N(s))$ recoding with $u_i = s_{i_3} s_{i_2} s_{i_1} s_{i_0} \overline{s}_{i-1_4} s_{i-1_3}$ as discussed above.

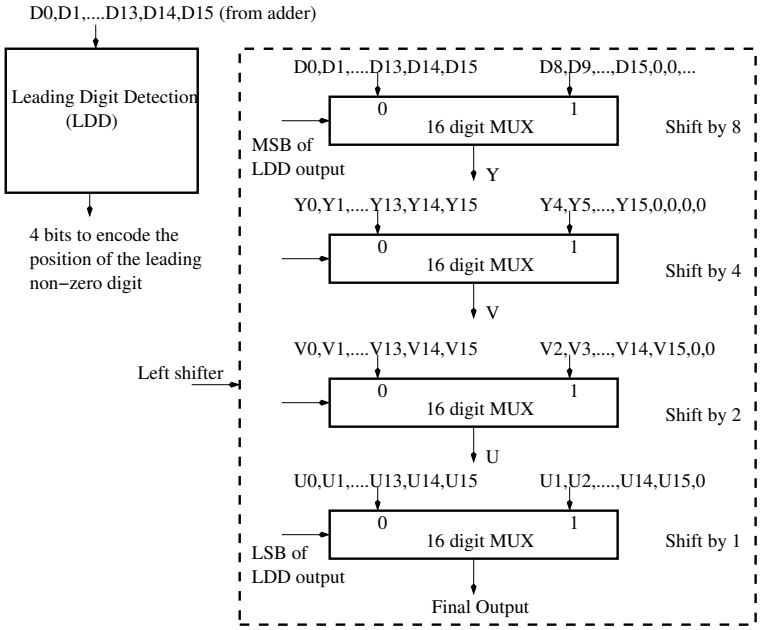


Figure A.3: Leading digit detection with a hierarchical tree approach.

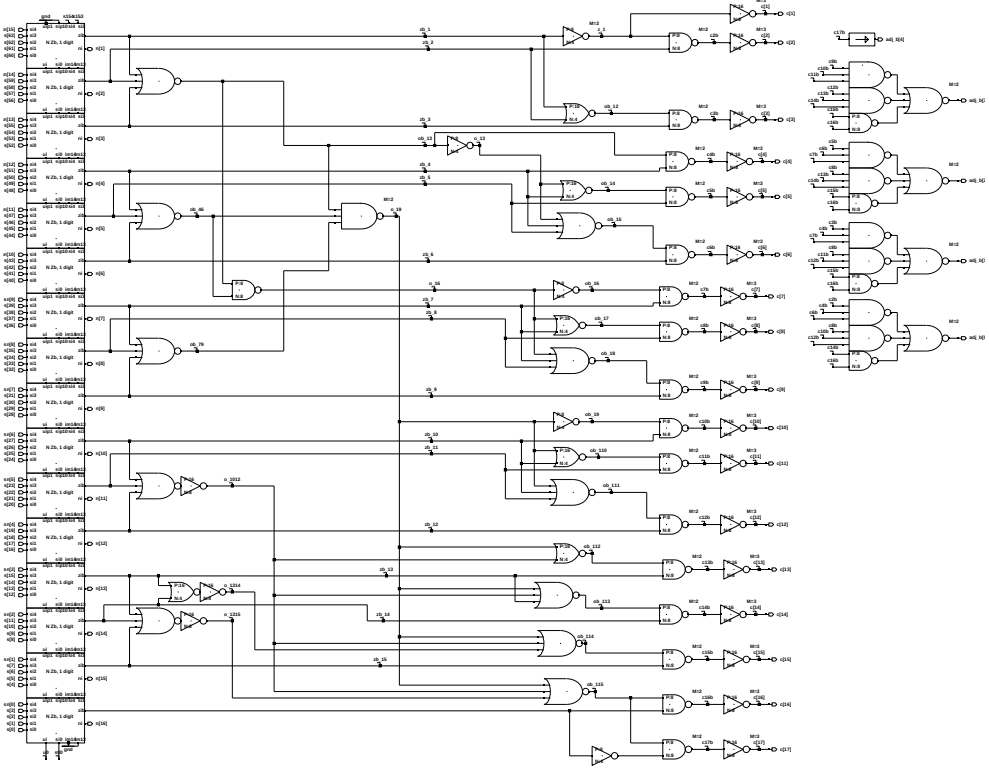


Figure A.4: Encoding the position of the leading digit with a priority encoder.

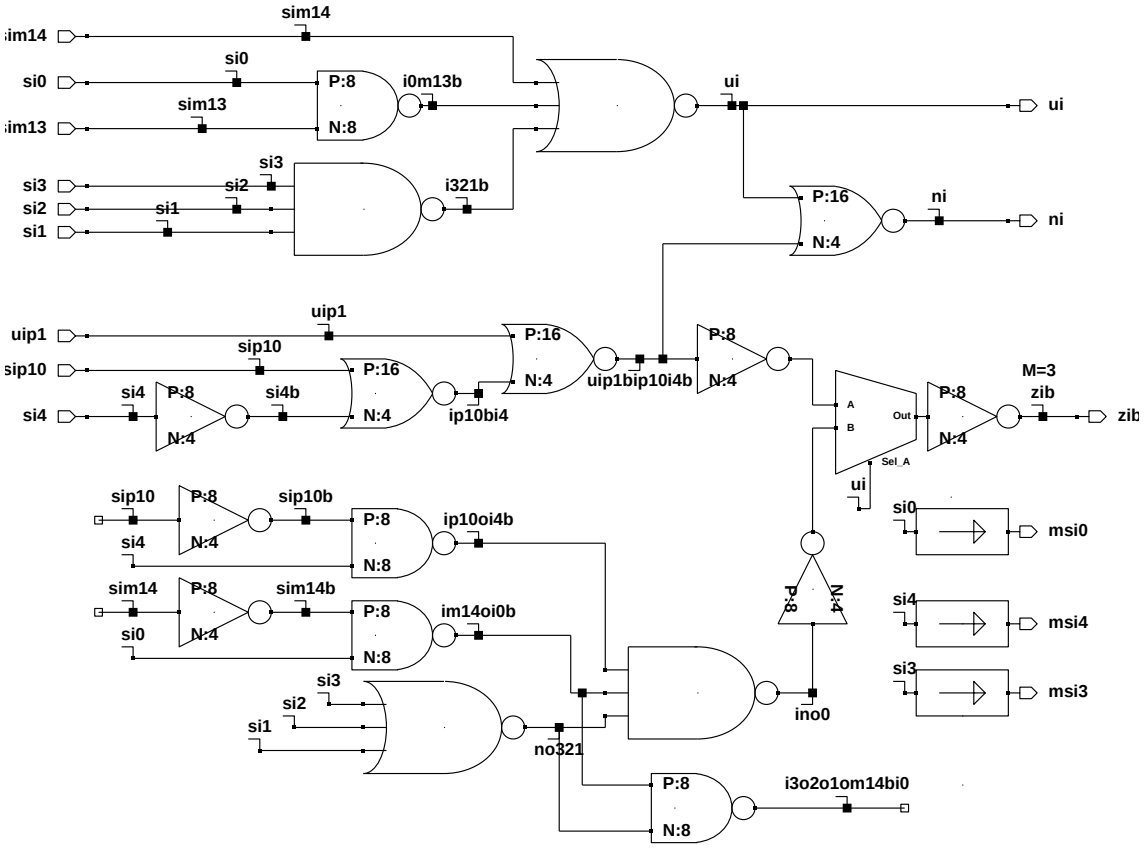


Figure A.5: Production of the n_i and z_i signals.

In conclusion, the recoding techniques discussed in this appendix eliminate the need for complex pattern matching schemes. That enables a simpler implementation for the leading digit detection block needed in the proposed redundant digit floating point adder design. The optimum choice of u_i to minimize the hardware complexity or to minimize the time delay of the LDD is an issue for further research.

Appendix B

Rounding logic

In the proposed system, the numbers are saved before rounding them to the register file. When such a number is used by another functional unit, it is rounded first. The position of the least significant bit of the number where the rounding takes place (the rounding location) is determined by the leading non-zero bit of the MSD and the signed sticky digit of the part below that leading bit. In the following sections, the hardware determining the rounding location and deciding on the rounding value is described. The rounding for the multiplier where three special correction bit vectors are added to the partial products is also explained.

B.1 Rounding in the adder

Fig. B.1 shows the rounding logic used in the adder given our definition of the bits of the MSD and LSD of the redundant format as

$$\begin{array}{cccc|ccc|cccc|ccc} a & b & c & d & \cdots & \cdots & \cdots & f & g & h & i & g_0 & r & s_0 \\ & & & & & & e & & & & g_1 & & s_1 & \end{array}$$

At the top left, simple logic gates use the bits of the MSD to determine the signals ld, lc, lb and la indicating the approximate leading bit. Once the signed sticky (ss) signal is known, the exact location is determined. The signed sticky is generated by another block that is not shown in this figure and it indicates whether the bits below the the leading one of the MSD constitute a positive or negative number. Hence there are five possibilities for the exact location of the leading bit: either one of the four bits of the MSD or one bit lower than d. This last cases occurs if $a = b = c = 0$, $d = 1$ and $ss = 1$. This is a case of fine adjustment as mentioned in the discussion regarding the leading digit detection. Let us call this case dm as a short for d minus one. If the leading bit is a then the location of rounding is at bit f of the LSD and so on for the other locations.

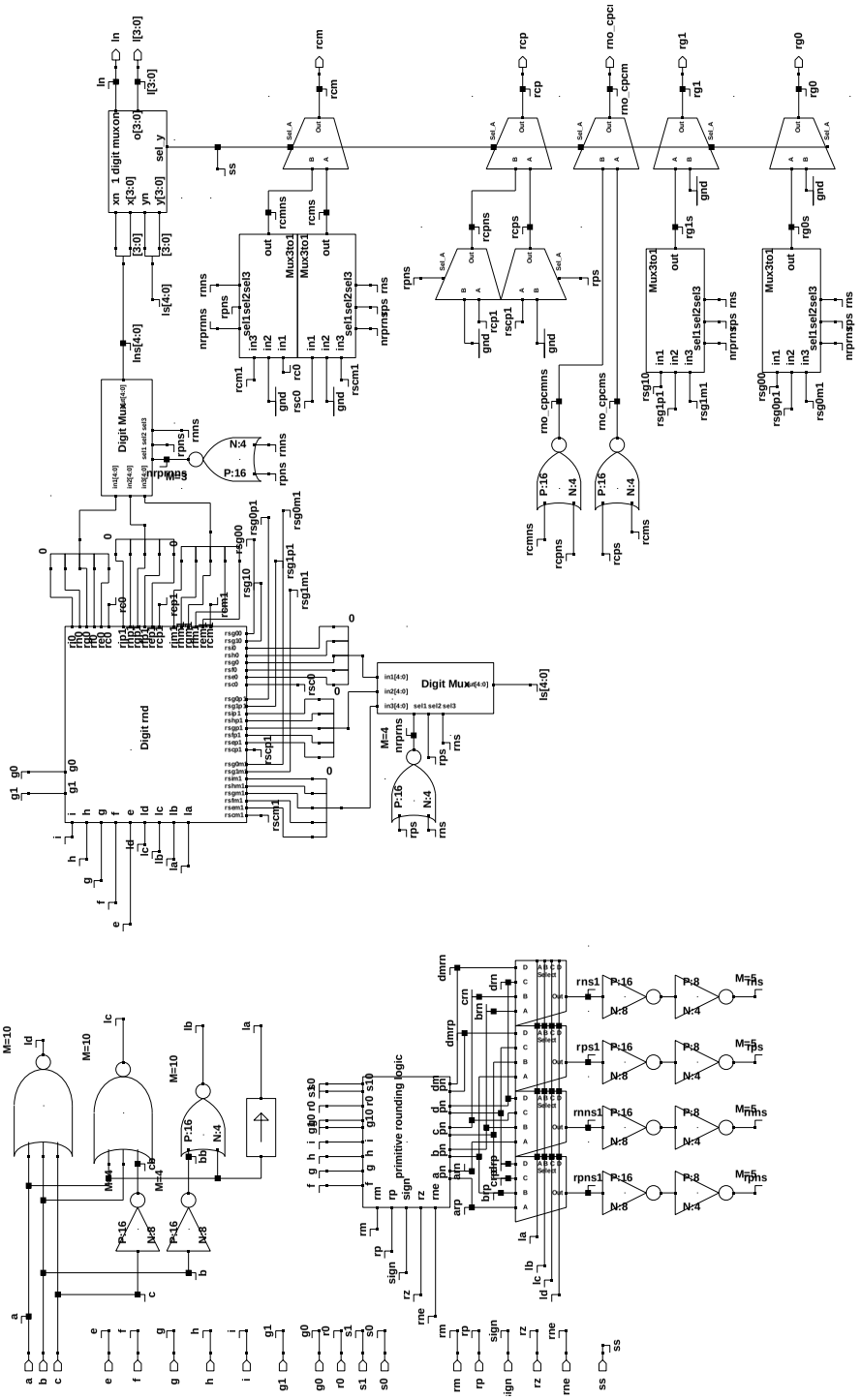


Figure B.1: Rounding of the least significant digit.

Table B.1: Logic equations for r_p and r_n

Leading bit	Round at	Rounding value
dm	g_0	$r_p = RNE\, r\overline{s}_1(s_0 + g_0)$
		$+ (RP\,\overline{sign} + RM\, sign)(r + \overline{s}_1 s_0)$
		$r_n = (RP\, sign + RM\,\overline{sign} + RZ)\overline{r}s_1$
d	i	$r_p = RNE\, \overline{g}_1 g_0 \overline{\overline{r}s_1}(r + s_0 + i)$
		$+ (RP\,\overline{sign} + RM\, sign)\overline{g}_1(g_0 + r + \overline{s}_1 s_0)$
		$r_n = RNE\, g_1(\overline{g}_0 + \overline{r}s_1 + \overline{r}\,\overline{s}_0 i)$
		$+ (RP\, sign + RM\,\overline{sign} + RZ)(g_1 + \overline{g}_0\overline{r}s_1)$
c	h	$r_p = RNE\, i\overline{(g_1 + \overline{g}_1\overline{g}_0\overline{r}s_1)}(g_0 + r + s_0 + h)$
		$+ (RP\,\overline{sign} + RM\, sign)(i + \overline{(g_1 + \overline{g}_1\overline{g}_0\overline{r}s_1)}(g_0 + r + s_0))$
		$r_n = (RP\, sign + RM\,\overline{sign} + RZ)\overline{i}(g_1 + \overline{g}_1\overline{g}_0\overline{r}s_1)$
b	g	$r_p = RNE\, h(i + \overline{(g_1 + \overline{g}_1\overline{g}_0\overline{r}s_1)})(i + g_0 + r + s_0 + g)$
		$+ (RP\,\overline{sign} + RM\, sign)(h + i + \overline{(g_1 + \overline{g}_1\overline{g}_0\overline{r}s_1)}(g_0 + r + s_0))$
		$r_n = (RP\, sign + RM\,\overline{sign} + RZ)\overline{hi}(g_1 + \overline{g}_1\overline{g}_0\overline{r}s_1)$
a	f	$r_p = RNE\, g(h + i + \overline{(g_1 + \overline{g}_1\overline{g}_0\overline{r}s_1)})(h + i + g_0 + r + s_0 + f)$
		$+ (RP\,\overline{sign} + RM\, sign)(g + h + i + \overline{(g_1 + \overline{g}_1\overline{g}_0\overline{r}s_1)}(g_0 + r + s_0))$
		$r_n = (RP\, sign + RM\,\overline{sign} + RZ)\overline{ghi}(g_1 + \overline{g}_1\overline{g}_0\overline{r}s_1)$

The *primitive rounding logic* block to the left of the figure is responsible for generating the speculative bits indicating the rounding value. The rounding value could be positive, negative or zero and hence is represented by two bits r_p and r_n and its value is given by $r_p - r_n$. Since the digits in the proposed format are always between -15 and $+15$, then the digit that was shifted out and from which the guard and round bits were drawn also falls in that range. Hence, if $g_1 = 1$ we are guaranteed that at least one other bit in that digit that was shifted out is one. So, either $g_0 = 1$, $r = 1$ or the sticky digit is positive (i.e. $s_1 = 0$ and $s_0 = 1$). This property insures that the fractional range at bit i the least significant bit of the LSD is bounded between -1 and 1. At g_0, this property insures that the fractional range is bounded between 0 and 1. For the other bit locations, the fractional range is bounded between -0.5 and 1. Given the location of the leading bit, the values of r_p and r_n are derived based on Table 4.1 resulting in the equations shown in Table B.1. Those equations insure that either $r_p = rn = 0$ or only one of them is set to one but never both at the same time.

The *Digit rnd* part in the middle of the figure generates three possible outcomes for the cases of positive, zero or negative rounding value at each possible rounding location. These outcomes are

Table B.2: Logic equations for the rounded LSD

Round at	$r_p - r_n$	co	e'	f'	g'	h'	i'	g'_1	g'_0
	$+1$	$\overline{e}fghi$	$e\overline{(fghi)}$	$f \oplus ghi$	$g \oplus hi$	$h \oplus i$	$\overline{i}$	0	0
i	0	0	e	f	g	h	i	0	0
	-1	$e\overline{f}\overline{g}\overline{h}$	$\overline{e}\overline{f}\overline{g}\overline{h}\overline{i} + e\overline{(\overline{f}\overline{g}\overline{h}i)}$	$\overline{f} \oplus (g+h+i)$	$\overline{g} \oplus (h+i)$	$\overline{h} \oplus i$	$\overline{i}$	0	0
	$+1$	$\overline{e}fgh$	$e\overline{(fgh)}$	$f \oplus gh$	$g \oplus h$	$\overline{h}$	0	0	0
h	0	$e\overline{f}\overline{g}\overline{h}$	$e(f+g+h)$	f	g	h	0	0	0
	-1	$e\overline{f}\overline{g}$	$\overline{e}\overline{f}\overline{g}\overline{h} + e\overline{(\overline{f}\overline{g}h)}$	$\overline{f} \oplus (g+h)$	$\overline{g} \oplus h$	$\overline{h}$	0	0	0
	$+1$	$\overline{e}fg$	$e\overline{(fg)}$	$f \oplus g$	$\overline{g}$	0	0	0	0
g	0	$e\overline{f}\overline{g}$	$e(f+g)$	f	g	0	0	0	0
	-1	$e\overline{f}$	$\overline{e}\overline{f}\overline{g} + e\overline{(\overline{f}g)}$	$\overline{f} \oplus g$	$\overline{g}$	0	0	0	0
	$+1$	$\overline{e}f$	$e\overline{f}$	$\overline{f}$	0	0	0	0	0
f	0	$e\overline{f}$	ef	f	0	0	0	0	0
	-1	e	$\overline{f}$	$\overline{f}$	0	0	0	0	0

the values of the bits in the locations of e to g_0 which we can call e' to g'_0 to signify the rounded values. In addition to that there might be a carry out (co) to the next higher digit. These values are given by the logic equations shown in Table B.2.

A few important points played a role in deriving those equations:

1. Since r_p and r_n are mutually exclusive, it is possible to use only one signal, co, to denote a positively valued carry in the case of $r_p = 1$ and a negatively valued carry otherwise.

2. For the rest of the units, it is important to guarantee that the generated digits are between -15 and $+15$. Hence, even in the case of truncation of the LSD there might be a carry out of it. This occurs if $e = 1$ and the other bits that were to remain are all zeros. In this case, a $co = 1$ signal is generated and e' is reset to zero.

3. The case of rounding at g_0 is slightly different.

   $r_p = 1$: Then $g'_0 = \overline{g}_0$, $g'_1 = g_1\overline{g}_0$ and a carry of $\overline{g}_1 g_0$ must be added to the LSD. A multiplexer with $\overline{g}_1 g_0$ as a select line is used to choose between the outcome of rounding at i when the rounding value is 1 or 0.

   $r_p = r_n = 0$: This is a case of truncation and, as above, we must guarantee that the new digit is within the permissible range. Hence, $g'_0 = g_0$ but $g'_1 = g_1 g_0$ and a negatively valued

carry of $g_1\overline{g}_0$ is added to the LSD. Again, a multiplexer with select line equal to $g_1\overline{g}_0$ is used to choose between the rounded LSD at location i when the rounding value is -1 or 0.

$r_n = 1$**:** Then $g'_0 = \overline{g}_0$, $g'_1 = \overline{g}_0$ and a negative carry of g_1 is added to the LSD. The select line of the multiplexer is then g_1 and the inputs are the rounded LSD at i when the rounding value is -1 or 0.

Since the signed sticky, ss, signal is a late signal ($\mathcal{O}(\log n)$ as discussed earlier) the *Digit rnd* block speculatively uses the approximate location of the leading bit to multiplex all the possible outcomes and to generate two sets of digits. A set for the case of a zero signed sticky and another set for the case of a signed sticky equal to one. As shown in Fig. B.1, the set assuming that the signed sticky is one enters into the multiplexer in the center of the figure that selects the correct digit depending on the values of r_p and r_n given that $ss = 1$. The other set is channeled to the multiplexer to the right of the *Digit rnd* block which also selects the correct digit depending on the values of r_p and r_n given that $ss = 0$. Finally, the multiplexer to the top right of the figure gives the final output digit depending on the value of ss. The signed sticky ss is also used as a select line for the other multiplexers to the right of the figure giving the values of the rounded g_1 and g_0 as well as the possible three values for the carry out to the next higher digit: $rcp = 1$ if it is positive, $rcm = 1$ if it is negative otherwise if it is zero the signal *rno-cpcm* is set to one.

Similar to the encoder of the LDD, the logic deciding the ss signal detects the sign of each digit and whether it is zero or not. Then, the priority encoder shown in Fig. B.2 is used to determine the value of ss which is given by the signal $N116$ while the signal $Z116$ indicates if all the bits of the significand are zeros (useful for checking for a zero input).

B.2 Rounding in the multiplier

As mentioned in section 5.2.2, the multiplication is divided into three parts:

$X \times Y =$	$X_{chopped}Y_{chopped}$	By Booth recoding, generating partial products and summing
	$+(b_x + r_x)Y_{chopped} + (b_y + r_y)X_{chopped}$	2 special partial products added in the tree
	$+(b_x + r_x)(b_y + r_y)$	special correction added to the tree

Fig. B.3 shows how $(b_y + r_y)X_{chopped}$ is formed.[1] The rounding portion $(b_y + r_y)$ is done by feeding the bits of Y to the correction block in the bottom right of the figure to generate five shifting signals sh_4 to sh_0 and the correction rounding value. The output of the correction rounding value

[1] Obviously, the same circuit with the inputs reversed can do $(b_x + r_x)Y_{chopped}$.

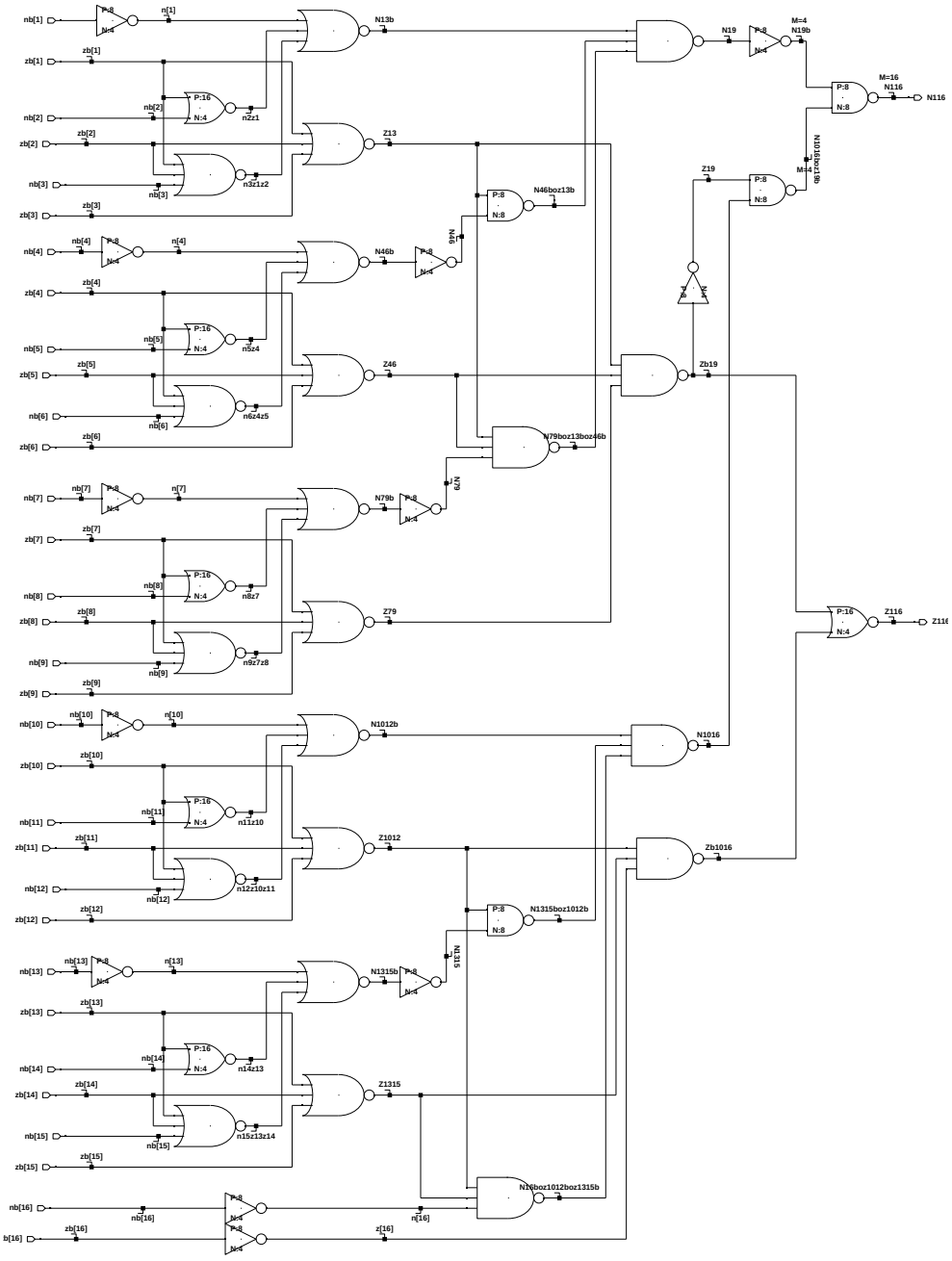


Figure B.2: Encoding the signed sticky digit using a priority encoder.

or is coded by two bits or_p and or_n and its value is $or_p - or_n$. In the figure, the complement of those two signals are shown as $orpb$ and $ornb$. This correction block at the bottom right basically recodes the multiplication by $(b_y + r_y)$ as a multiplication by 2^{shift} followed by a selection of either the positive or negative of the shifted operand based on the value of the or_p and or_n signals.

The correction block must decide on the exact amount of shifting to be done. For this to be achieved, the leading one of the MSD of Y is detected. Similar to the rounding in the adder, let us assume that the bits of the MSD are a b c d and let us assume that the signals indicating which bit is the leading one are labeled la, lb, lc and ld. Those signals are among the inputs to the correction block. The other inputs are the value of the signed sticky ss, the rounding mode selectors, the sign of the operand and the bits of the LSD and guard, round and sticky digits of Y. As in the discussion of the adder, let us define the bits of Y as:

$$\begin{array}{cccc|ccc|cccc|ccc} a & b & c & d & \cdots & \cdots & \cdots & f & g & h & i & g_0 & r & s_0 \\ & & & & & & e & & & & g_1 & & s_1 & \end{array}$$

If the signed sticky is not set ($ss = 0$), the value of b_y the bit directly after the rounding location is irrelevant, the output corrected value or is given by r_y alone and the shifting amount is determined only by the leading one of the MSD. On the other hand, if $ss = 1$ then b_y must be considered. The evaluation of $(b_y + r_y)$ is tricky since r_y could be negative, zero or positive while b_y is positive or zero except at the g_0 location where it might be negative, zero or positive. Hence, when $ss = 1$ and the rounding is

not at the g_0 location: If both b_y and r_y are positive then $(b_y + r_y) = 2$ which means that, effectively, it is carried to the next higher bit location and the shift amount is increased by one canceling the effect of ss. Otherwise, the effect of ss is taken into account and the shift amount derived based on it.

at the g_0 location: The value of $g_1 g_0$ is given by $-2g_1 + g_0$ and can be in $\{-2, -1, 0, 1\}$

- $g_1 g_0 = 01 = (1)$ then, as above, if $r_y = 1$ it is a carry to the next higher bit. Otherwise, the effect of ss is taken into account.
- $g_1 g_0 = 00 = (0)$ then $b_y = 0$ and the effect of ss is taken into account.
- $g_1 g_0 = 11 = (-1)$ then if $r_y = -1$ (note the sign) it is a carry to the next higher bit. Otherwise, the effect of ss is taken into account.
- $g_1 g_0 = 10 = (-2)$ in this case $r_y \in \{0, 1\}$[2] and if it is zero a carry to the next higher bit occurs. Otherwise, the effect of ss is taken into account.

[2]Due to the condition on the number system that a digit cannot be -16, if $g_1 g_0 = 10$ then in the bits representing the guard round and sticky digits either the round bit $r = 1$ or the sticky digit is positive. This leads to only a positive fractional value at g_0 which means that the rounding value at this location cannot be negative as noted in section B.1.

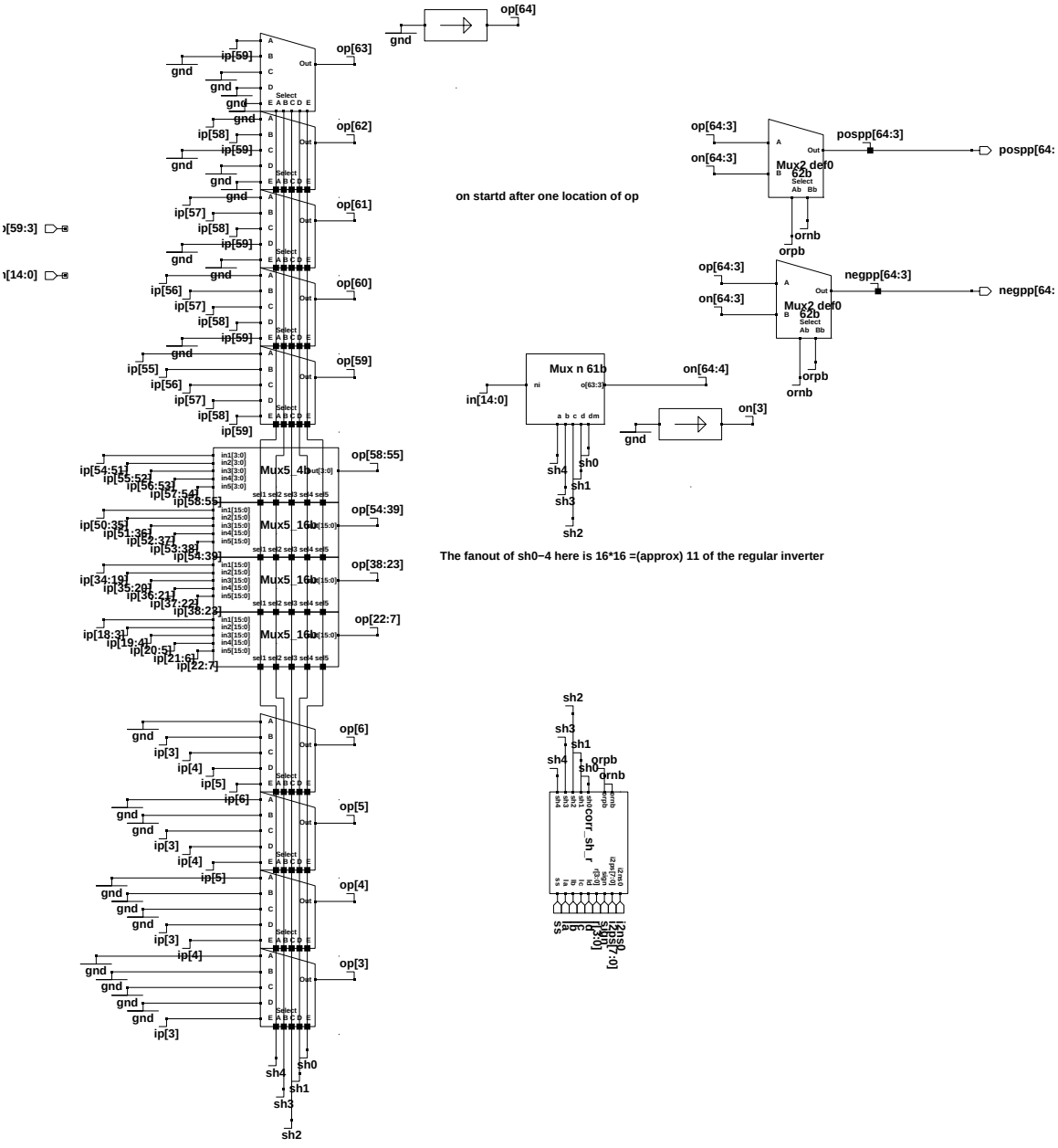


Figure B.3: Multiplying one operand by the rounding part of the other.

Since the rounding location could be anywhere from the g_0 location to the f location, the shift amount produced by the correction block is assumed to be zero if the value of $(b_y + r_y)$ evaluates to be at the g_0 location. Let the rounding values at any location be represented by r_p and r_n with a subscript referring to the leading bit of the MSD corresponding to this location. These rounding values are evaluated by the *primitive rounding logic* block described in the discussion of Fig. B.1. The signals for the shift are then given by the following equations:

$$\begin{aligned}
sh_4 &= \overline{ss}\, la + ss\, la(gr_{p_b}) \\
sh_3 &= \overline{ss}\, lb + ss\, la\overline{(gr_{p_b})} + ss\, lb(hr_{p_c}) \\
sh_2 &= \overline{ss}\, lc + ss\, lb\overline{(hr_{p_c})} + ss\, lc(ir_{p_d}) \\
sh_1 &= \overline{ss}\, ld + ss\, lc\overline{(ir_{p_d})} + ss\, ld(\overline{g}_1 g_0 r_{p_{dm}} + g_1 g_0 r_{n_{dm}} + g_1 \overline{g}_0 \overline{r}_{p_{dm}}) \\
sh_0 &= ss\, ld\, \overline{(\overline{g}_1 g_0 r_{p_{dm}} + g_1 g_0 r_{n_{dm}} + g_1 \overline{g}_0 \overline{r}_{p_{dm}})}
\end{aligned}$$

The output corrected rounding value or corresponds to the value of $b_y + r_y$ regardless of the shift. So, $or = 0$ if $b_y + r_y = 0$, $or = 1$ if $b_y + r_y = 1$ or 2 and $or = -1$ if $b_y + r_y = -1$ or -2. To calculate or_p and or_n, we can use two intermediary signals r_p and r_n indicating the value of r_y,

$$\begin{aligned}
r_p &= \overline{ss}(la\, r_{p_a} + lb\, r_{p_b} + lc\, r_{p_c} + ld\, r_{p_d}) + ss(la\, r_{p_b} + lb\, r_{p_c} + lc\, r_{p_d} + ld\, r_{p_{dm}}) \\
r_n &= \overline{ss}(la\, r_{n_a} + lb\, r_{n_b} + lc\, r_{n_c} + ld\, r_{n_d}) + ss(la\, r_{n_b} + lb\, r_{n_c} + lc\, r_{n_d} + ld\, r_{n_{dm}})
\end{aligned}$$

The signals for or_p and or_n are mutually exclusive to fit their role as multiplexer select signals in Fig. B.3. The signal or_p is set to 1 if $r_y = 1$ and b_y is neglected in the case of $ss = 0$ or $b_y = 0$. The other case where or_p is set is if r_y is not a negative one and $b_y = 1$. In boolean logic this translates to:

$$\begin{aligned}
or_p &= \overline{ss}\, r_p + ss\, r_p(la\, \overline{g} + lb\, \overline{h} + lc\, \overline{i} + ld\, \overline{g}_1 \overline{g}_0) \\
&\quad + ss\, \overline{r}_n(la\, g + lb\, h + lc\, i + ld\, \overline{g}_1 g_0)
\end{aligned}$$

Similarly, or_n is set to 1 if $r_y = -1$ and b_y is neglected in the case of $ss = 0$ or $b_y = 0$. The second case where or_n is set is if r_y is not a positive one and $b_y = -1$ ($g_1 g_0 = 11$) or $b_y = -2$ ($g_1 g_0 = 10$). The third case is if $r_y = 1$ and $b_y = -2$. In boolean logic this translates to:

$$\begin{aligned}
or_n &= \overline{ss}\, r_n + ss\, r_n(la\, \overline{g} + lb\, \overline{h} + lc\, \overline{i} + ld\, \overline{g}_1 \overline{g}_0) \\
&\quad + ss\, \overline{r}_p\, ld\, g_1 \\
&\quad + ss\, r_p\, ld\, g_1 \overline{g}_0
\end{aligned}$$

With the values of sh_4 to sh_0, or_p and or_n the multiplexers of Fig. B.3 shift and choose the

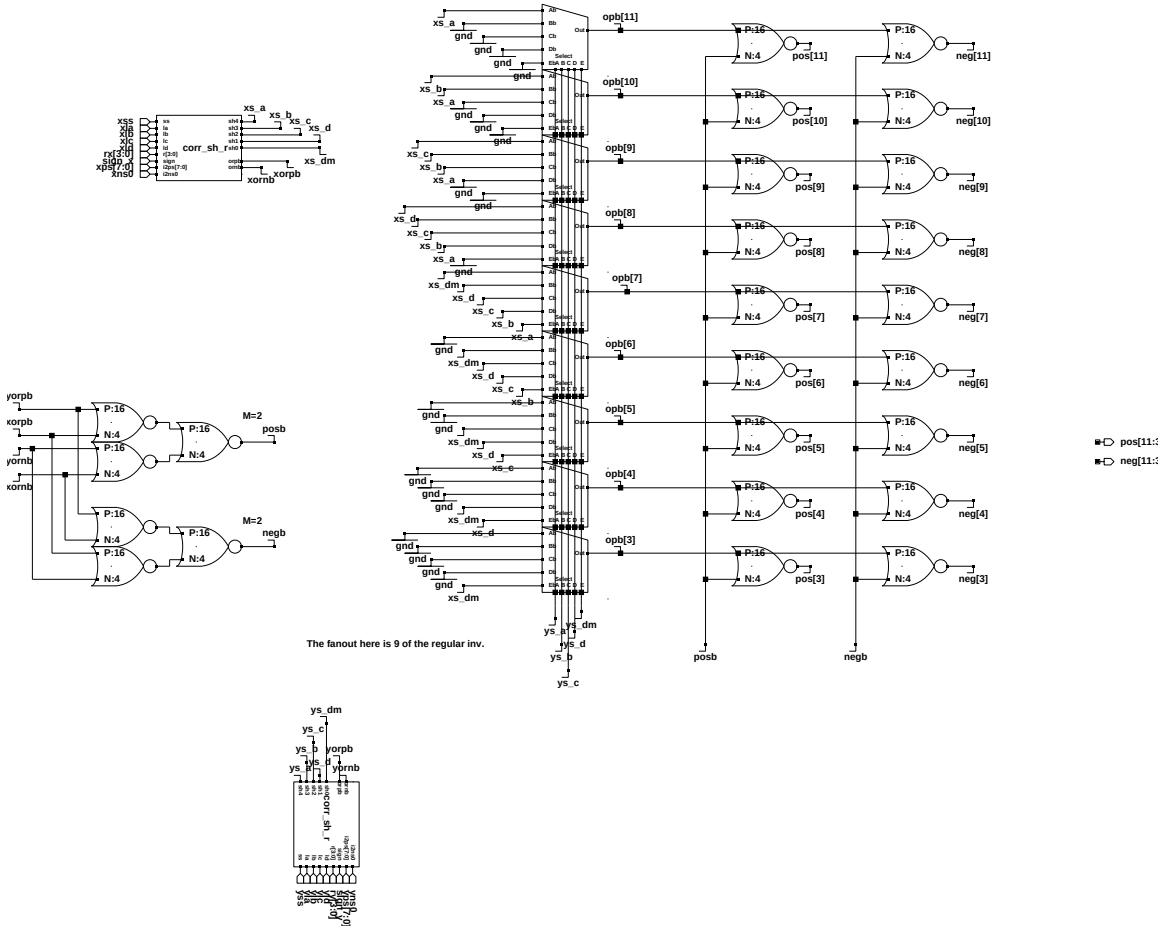


Figure B.4: Multiplying the rounding parts of the two operands.

correct positive and negative outputs as seen in the top left part of the figure.

The calculation of the special correction $(b_x + r_x)(b_y + r_y)$ is done in a similar fashion as shown in Fig.B.4. The same correction block described above is used at the top left for operand X and at the bottom left for operand Y. The logic gates in the center left of the figure determine that the result is positive if the output corrected rounding values of both X and Y are of the same sign. The result is negative if those values are of opposite sign. The multiplexers followed by the *NOR* gates at the right then set the correct bit.[3]

[3]Those multiplexers have complemented outputs, that is why *NOR* gates are used.

B.3 Rounding conclusions

Due to the redundancy, the rounding location may shift by one bit depending on the signed sticky of the part below the leading bit of the number being rounded. Since the generation of this signed sticky depends on all the bits of the number, it is slow. Special attention is given to optimize the speed of the hardware given the slow signed sticky by speculatively calculating different possibilities for the rounded LSD. Once the exact rounding location is determined the corresponding rounded LSD is chosen. For the multiplier, the generation of the rounding correction is explained and an efficient implementation is presented.

Appendix C

Implementation details of the adder

The proposed design with the four IEEE rounding modes is implemented at the transistor level using a scalable CMOS technology with $n = 53$, $f = 3$ and $r = 4$.

The circuits mostly use static CMOS technology gates with only few parts using NMOS pass transistors (namely the shifters). The schematic entry tool *sue* is used to describe the circuit connections and the design is simulated for functionality at the logic level using *verilog* and for speed at the transistor level using the switch level simulator *irsim*.

Since the signed digit addition is a fundamental component included in a number of the other parts, its implementation is described first followed by the details of all the floating point adder.

C.1 SD adder block

Let the i-th digits of the two input operands be denoted by x_i and y_i. For each position i, the position sum p_i, the carry out of this position c_i, the intermediate sum w_i and the final sum s_i are calculated according to the following rules to insure an addition free from carry-propagation [26, 27, 28]:

$$
\begin{aligned}
p_i &= x_i + y_i \\
c_i &= \begin{cases} -1 & \text{if } p_i \leq -\alpha \\ 1 & \text{if } p_i \geq \alpha \\ 0 & \text{otherwise} \end{cases} \\
w_i &= x_i + y_i - \beta c_i \\
s_i &= w_i + c_{i-1}
\end{aligned}
$$

Using $\beta = 16$ and $\alpha = 15$, the ranges for these different quantities are:

$$\begin{aligned} x_i, y_i, s_i &\in \{-15, \cdots, 15\} \\ p_i &\in \{-30, \cdots, 30\} \\ w_i &\in \{-14, \cdots, 14\} \end{aligned}$$

To represent the range of digits chosen, x_i and y_i are both 5 bits and p_i is formed using a normal 5 bit two's complement binary adder. If x_i and y_i are of different signs then their addition cannot generate a number larger in magnitude than 15 and no carry would be generated except if the magnitude of the result is exactly 15. If they are both positive, a carry of 1 is generated if the two's complement adder overflows (sum greater than 15) or if the sum is exactly 15. In this case of overflow, both x_i and y_i have their MSB 0 and the MSB of p_i is 1. To form w_i from p_i, we need to subtract $\beta \times c_i = 16 \times 1$. This is equivalent to inverting the MSB of p_i from 1 to 0. If p_i is exactly 15 (01111 in binary), then subtracting 16 to form w_i results in $w_i = -1$ (11111 in binary). Again, the difference between w_i and p_i is just in the complemented MSB. Similarly for the case where both x_i and y_i are negative, if an overflow occurs or if p_i is equal to exactly -15 (10001 in binary), then w_i is the same as p_i except for a complemented MSB. If α is chosen to be less than $\beta - 1$ then comparisons with more numbers ($15, 14, \cdots, \alpha$ and $-15, -14, \cdots, -\alpha$) would be required and a more involved scheme must be used for the calculation of w_i. This is why the decision is to use $\alpha = \beta - 1 = 15$.

A five bit adder is thus used to calculate p_i. Let us assume that the bits representing p_i are $p_{i_4}, p_{i_3}, \cdots, p_{i_0}$ and that the carry into the MSB of this adder is c_4 and the carry out of it is c_5. Further, c_i can be represented by two bits; $c_{i(1)} = 1$ if $c_i = 1$ and $c_{i(-1)} = 1$ if $c_i = -1$, if $c_i = 0$ then both bits are set to 0. Similar to p_i, w_i is represented by 5 bits. Then, s_i could be calculated as $w_i + 00001$ if $c_{i(1)} = 1$ or $w_i + 11111$ if $c_{i(-1)} = 1$ or simply as w_i when both are zero. Instead of making this two steps process (first w_i then s_i), it is worth noting that

$$\begin{aligned} s_i &= w_i + c_{i-1} \\ &= (p_i - rc_i) + c_{i-1} \\ &= (p_i + c_{i-1}) - rc_i \end{aligned}$$

This means that the correction needed to get w_{i_4} from p_{i_4} can be delayed till after adding the carry in of ± 1. We can thus have $x_i + y_i + 1$ and $x_i + y_i - 1$ calculated in parallel with $x_i + y_i$ and use a multiplexer with select lines depending on the carry into the digit. After the selection, the MSB of the result could be conditionally inverted to get the correct s_{i_4} depending on the carry out of the digit. Otherwise, the inversion can be made first and then the multiplexer is used. This latter method is what is actually used in the implementation as presented in Fig. C.1 where the

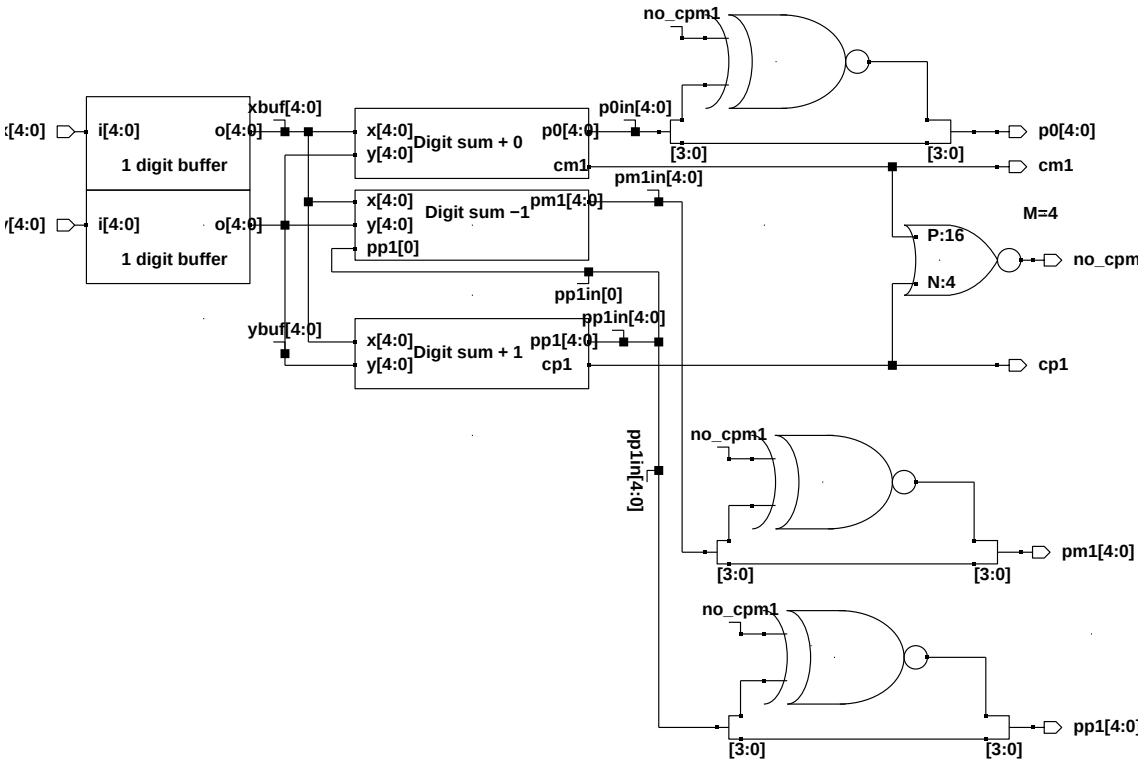


Figure C.1: Calculation of the possible outcomes for one digit addition.

signal *no-cpm*1 is an indicator that no carries occurred. If this indicator is low (meaning that a carry occurred), it insures that the most significant bit of the three outputs is inverted.

The boolean equations governing $c_{i(1)}, c_{i(-1)}$ are derived based on a few facts:

1. $c_{i(1)}$ is set in case of a positive overflow or a result equal to $+15$. Both of these can only occur if $x_{i_4} = y_{i_4} = 0$.
   - The adder calculating $x_i + y_i + 1$ has a positive overflow if $x_i + y_i \geq 15$.
2. $c_{i(-1)}$ is set in case of a negative overflow or a result equal to -15.
   - A zero input is represented by the pattern 00000 which is considered positive and an output of -15 can result due to one input being -15 and the other zero or both inputs negative and their sum equal to -15.
   - With the current choice of number system the case of one of the inputs being -16 is a don't care condition.

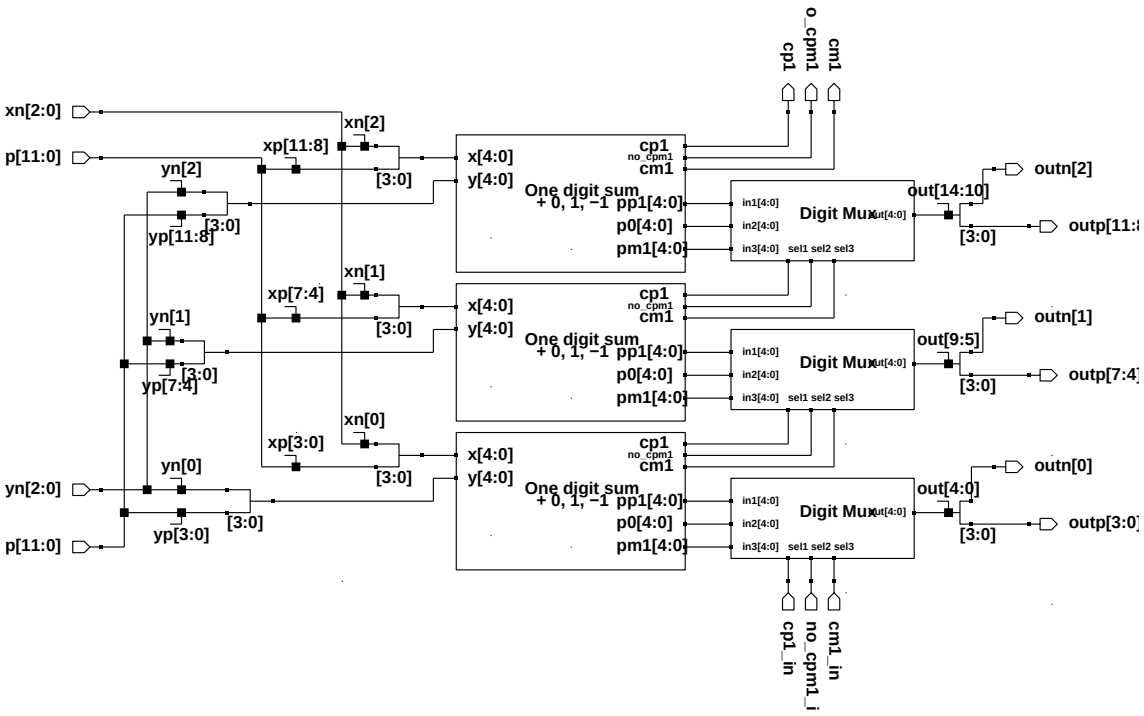


Figure C.2: Three consecutive digits in a signed digit adder.

Hence, if c'_4 is the carry into the MSB of the adder calculating $x_i + y_i + 1$ then

$$c_{i(1)} = \overline{x}_{i_4}\overline{y}_{i_4}c'_4$$

As for $c_{i(-1)}$, its equation should have three terms. The first one checks for a negative overflow when both inputs are negative ($x_{i_4} = y_{i_4} = 1$). The second term checks if the sum is exactly -15 and either both inputs are negative or at least one of them is negative. The third term checks if the sum is exactly -16 and either both inputs are negative or at least one of them is negative (allowing for the don't care case of a $-16 + 0$). Hence the equation becomes:

$$c_{i(-1)} = x_{i_4}y_{i_4}\overline{p}_{i_4} + (x_{i_4} + y_{i_4})p_{i_4}\overline{p}_{i_3}\overline{p}_{i_2}\overline{p}_{i_1}p_{i_0} + (x_{i_4} + y_{i_4})p_{i_4}\overline{p}_{i_3}\overline{p}_{i_2}\overline{p}_{i_1}\overline{p}_{i_0}$$

which could obviously be simplified further and written in terms of the individual bits of x_i and y_i to improve the time delay.

In Fig. C.1, $c_{i(1)}$ is calculated within the block evaluating $x_i + y_i + 1$ and shown as the signal named $cp1$ while $c_{i(-1)}$ is calculated in the block evaluating $x_i + y_i$ and shown as the signal named $cm1$. With these carries, a multi-digit adder can be built as shown in Fig. C.2 and this block can be used to build even bigger structures in the floating point adder.

C.2 Details of the floating point adder

The general view of the floating point adder is shown in Fig. C.3 while Fig. C.4 shows the cancellation path. The adder at the top of Fig. C.4 produces four results $A - B$, $A - shift(B)$, $B - A$ and $B - shift(A)$. Each of those is represented by two buses one for the extra negative valued bits and one for the positive bits. Depending on the speculation from the low order bits of the exponent, either the shifted version or the non-shifted version is chosen using the 16 digits multiplexers. The LDD takes its input from the $A - B$ branch and decides on the normalization shift amount. That amount is forwarded to two shifters, one for the $A - B$ branch and one for the $B - A$. The LDD forwards the shift amount also to the block deciding the exponent of the output in order to subtract the shift amount from the exponent of A.

Rounding is done in the cancellation path in parallel with the subtraction step. The rounding of the proposed format does not propagate a carry through the whole number as the rounding in conventional adders do. SD addition is used instead and the addition of a ± 1 digit representing the rounding decision is easily handled. The adder of the cancellation path is detailed in Fig. C.5. The two operands in this figure are labeled as x and y and the upper half of the figure produces $x - shift(y)$ and $x - y$ while the lower half produces $y - shift(x)$ and $y - x$. On the left, two blocks are responsible for correctly rounding the least significant digit of each of the two operands. This rounding circuit is the same circuit explained earlier in section 4.2.2. The carry out of this circuit to the next significant digit from both operands is forwarded to the block in the middle of the figure labeled $cx - cy$. This block calculates the mathematical value of those carries when subtracted from each other giving a result going from -2 to $+2$ as presented in Fig. C.6. That result is then provided to the two circuits calculating $x - y$ and $y - x$ to the right of Fig. C.5.

In the $x - y$ block, the rounded least significant digits are subtracted as shown in the bottom of Fig. C.7. According to the SD addition rules the least significant digit subtraction can generate a carry to affect the next higher digit. It is this next higher digit as well that receives the result from the block $cx - cy$ of Fig. C.6 calculating the difference of the carries. Hence, in the subtraction of the next higher digit, the result of the subtraction plus or minus up to three is produced. Then, the result of Fig. C.6 (difference of the carries) as well as the carries generated from the least significant digit stage are used to choose the correct answer. For the rest of the digit positions only up to plus or minus one of the result is produced and the corresponding carries are used to choose the correct result.

Going back to Fig. C.4, we see that the speculative signal *diff1* indicating an exponent difference of one is an output of the cancellation path. A block is located at the bottom of the path to use the output of the LDD and decide if a cancellation occurred or not. This block also decides on the sign of the result and whether it is an exact zero. All these signals are forwarded with the results of the cancellation path to the multiplexer choosing between the far and cancellation paths as shown in Fig. C.8.

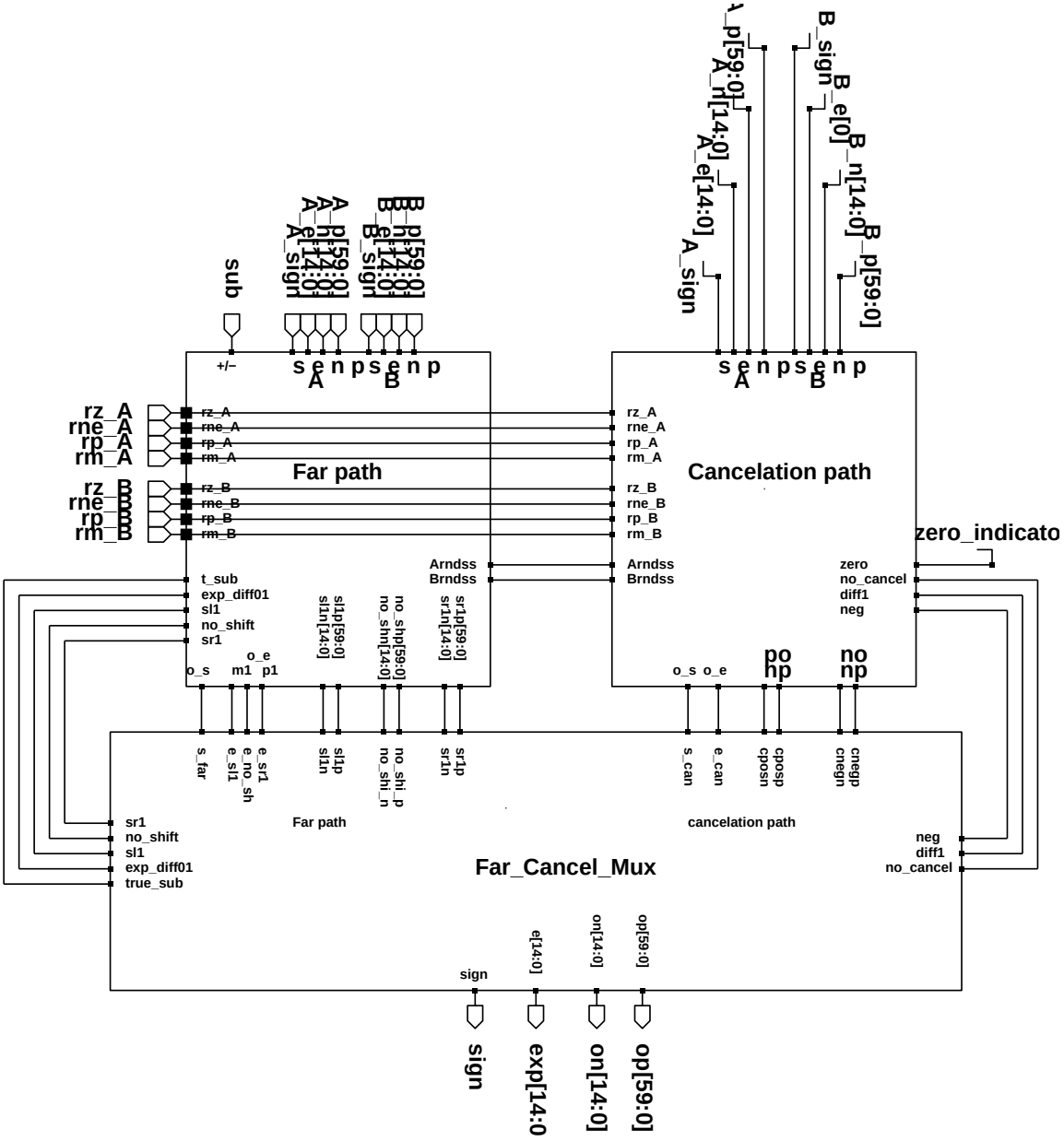


Figure C.3: General view of the floating point adder.

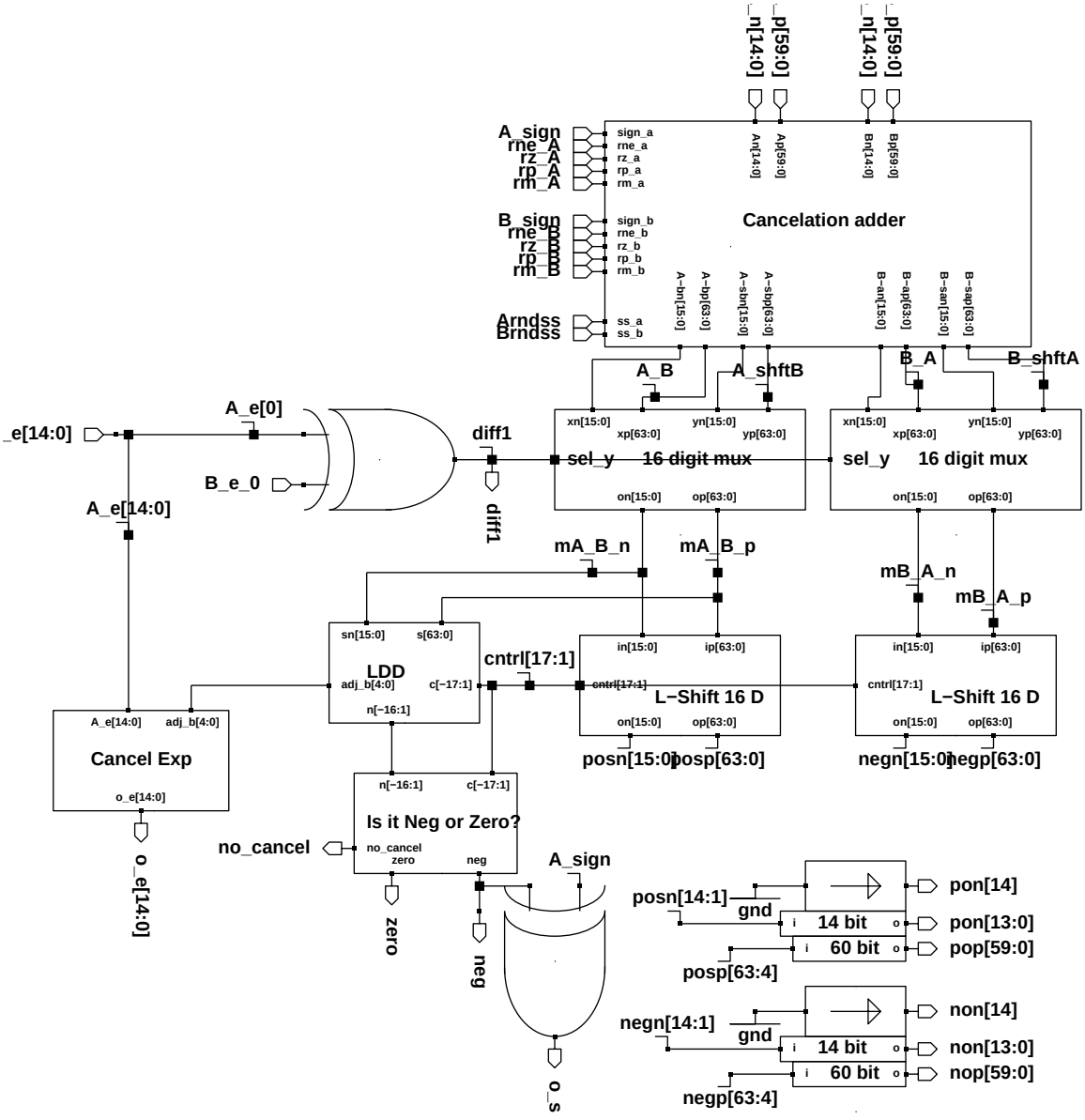


Figure C.4: Cancellation path of the floating point adder.

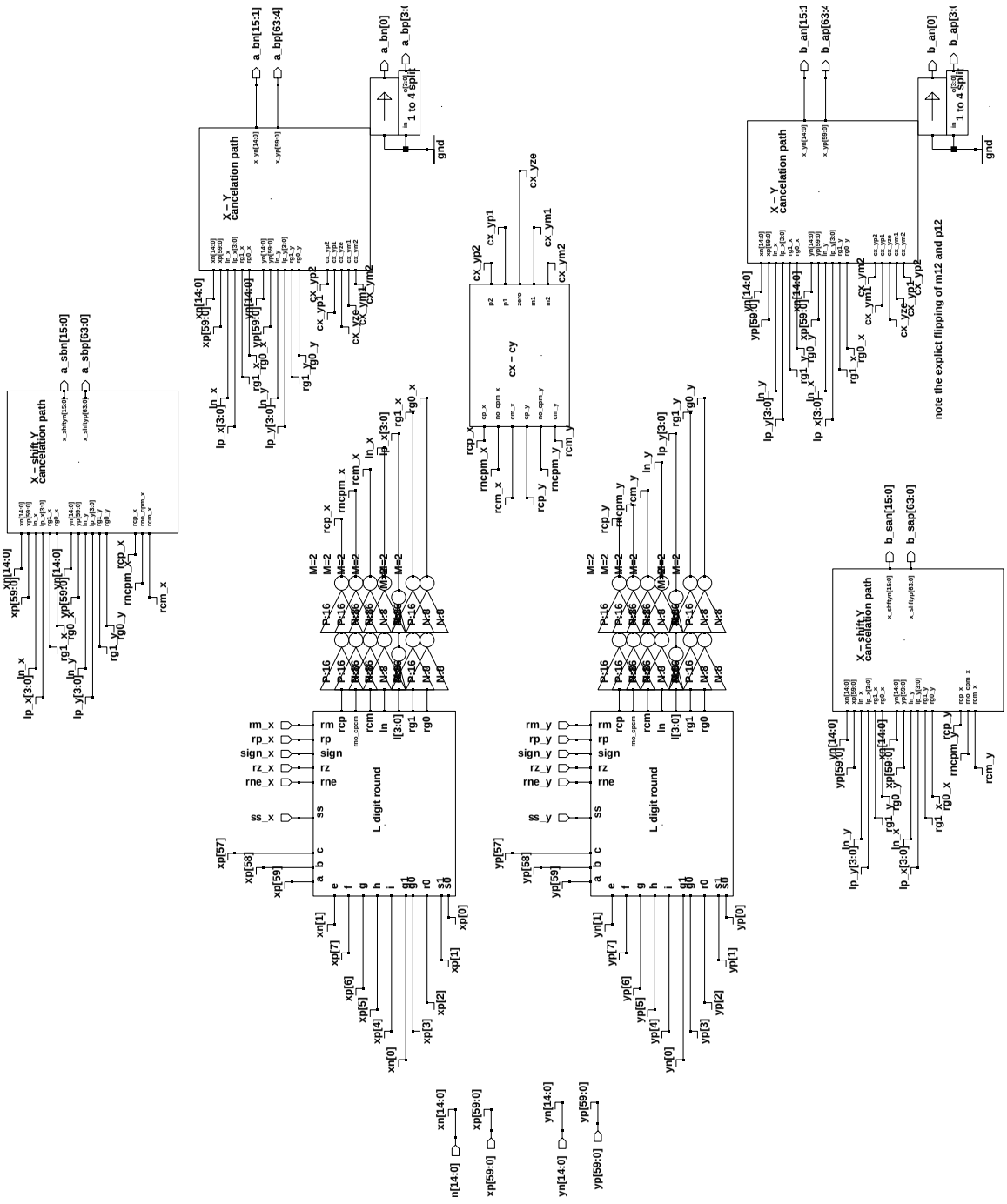


Figure C.5: The adder of the cancellation path.

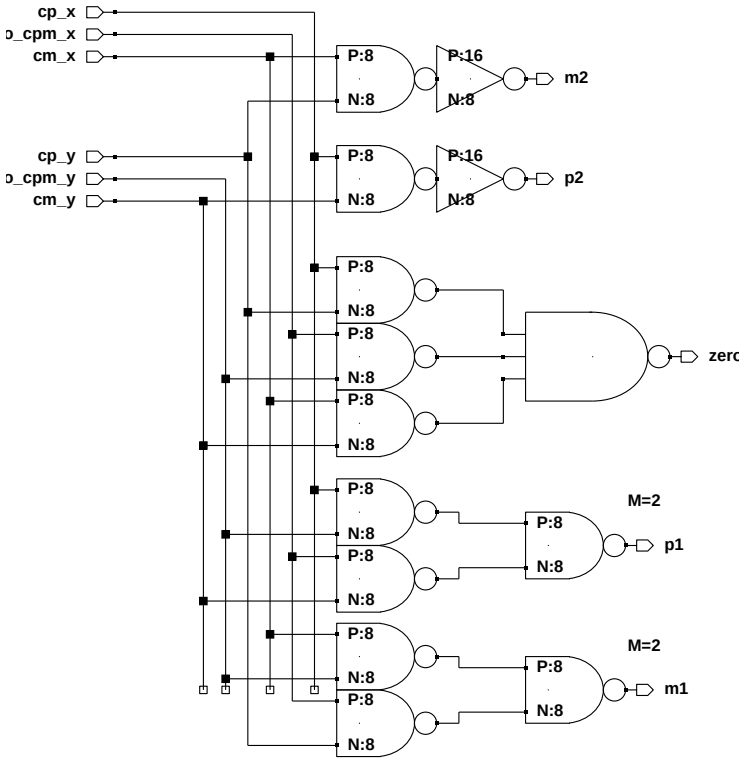


Figure C.6: Subtracting the carries in the cancellation adder.

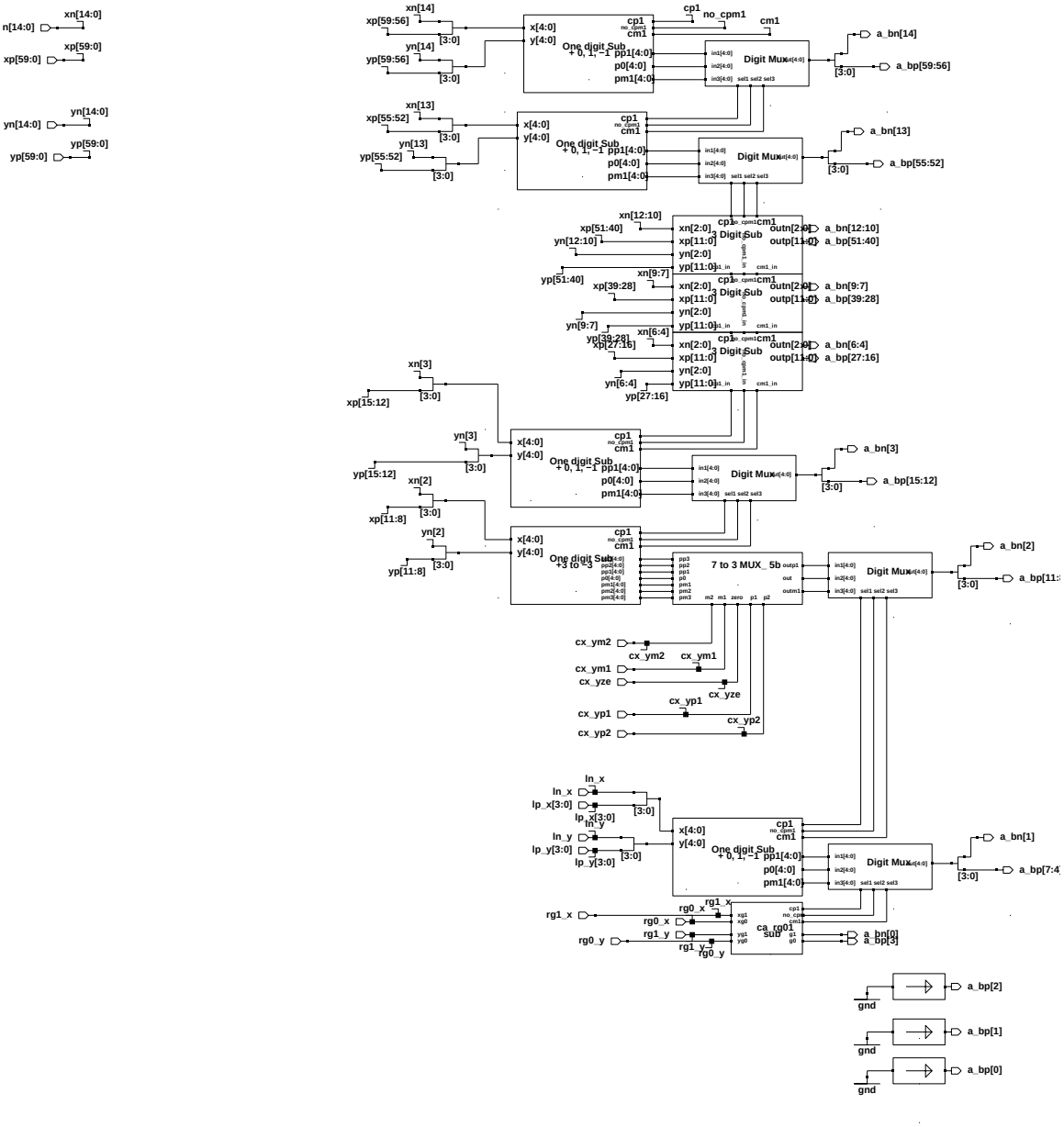


Figure C.7: Subtracting the operands in the cancellation adder.

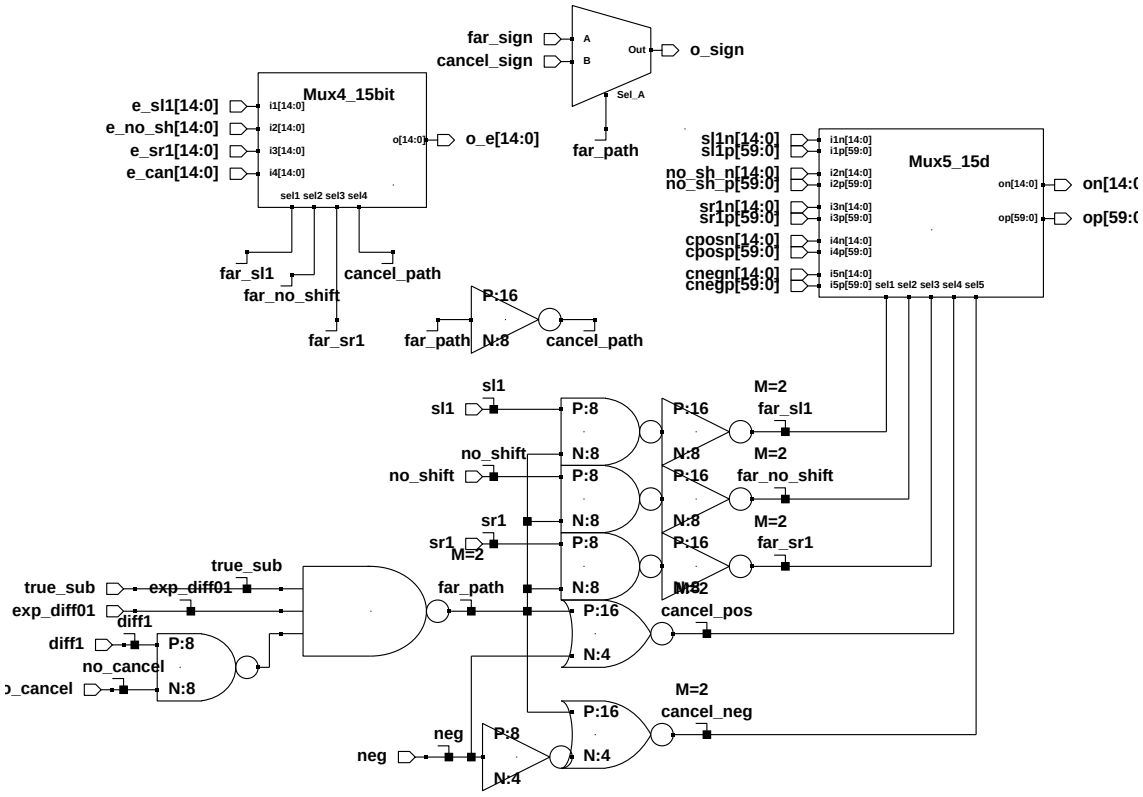


Figure C.8: Multiplexer choosing between the far and cancellation paths.

The logic at the bottom of Fig. C.8 indicates that the cancellation path is chosen if all the following conditions are satisfied:

1. It is a true subtraction.
2. The exponent difference is zero or one.
3. A cancellation occurred or the exponent difference is not one (i.e. it is zero.)

These are the same conditions stated in section 4.2 that allow the cancellation path to be free from any logic to produce the guard and sticky digits. If the cancellation path is chosen then depending on the sign of the result in that path (indicated by the signal *neg*) either the positive or the negative result is enabled.

Due to normalization, the result of the far path can be shifted left or right by one digit location. The far path produces the three possible outputs and if the far path is chosen one of these outputs is enabled as seen in Fig. C.8. The significand multiplexer in the figure is then a five to one multiplexer that chooses the correct output.

The schematic of the far path is shown in Fig. C.9. At the top of the figure, the signed sticky of the two operands is evaluated and then used in the blocks for rounding.

Two rounding blocks are used to simultaneously negate their outputs. Each operand is actually rounded and negated so that both the positive and negative alternatives are available to choose from in case of an effective subtraction operation indicated by the signal *t-sub*. The difference between "positive" and "negative" rounding blocks is shown in Fig. C.10. Basically, each digit is negated and the rounding of the least significant digit also produces the negative value. In this manner both rounded operands and their negatives are available by the time the exponent difference and multiplexer selection signals are ready. If, in another implementation, the exponent difference calculation takes a long time, the alignment shifting could also be overlapped with it in order to decrease the delay. In such a case two shifters will be needed and the choice of the "smaller" operand will be done after the shifting.

In the implementation at hand, depending on the sign of the exponent difference, one of the operands is chosen to be shifted. However, it is only the least significant bits of the difference that really determine the shift amount. If the shift amount is larger than the width of the operands then there is no need to shift the lesser operand and a zero can be added to the larger operand. The detection of this condition is done by the block labeled $g15$. In the presented scheme, only the least 4 bits are enough to represent a shift amount of 14 digits (the width of the operand). The subtracter of the exponents takes the two operand's exponents consisting of 15 bits each and computes $A_e - B_e$ by adding the two's complement of B_e to A_e. In parallel to that a second subtracter works on 4 bits only to produce the least significant part of $B_e - A_e$ which is then fed to a multiplexer with the least significant part of $A_e - B_e$. Since the two exponents are unsigned numbers, it is the carry out signal that determines the sign of the difference according to two's complement number addition

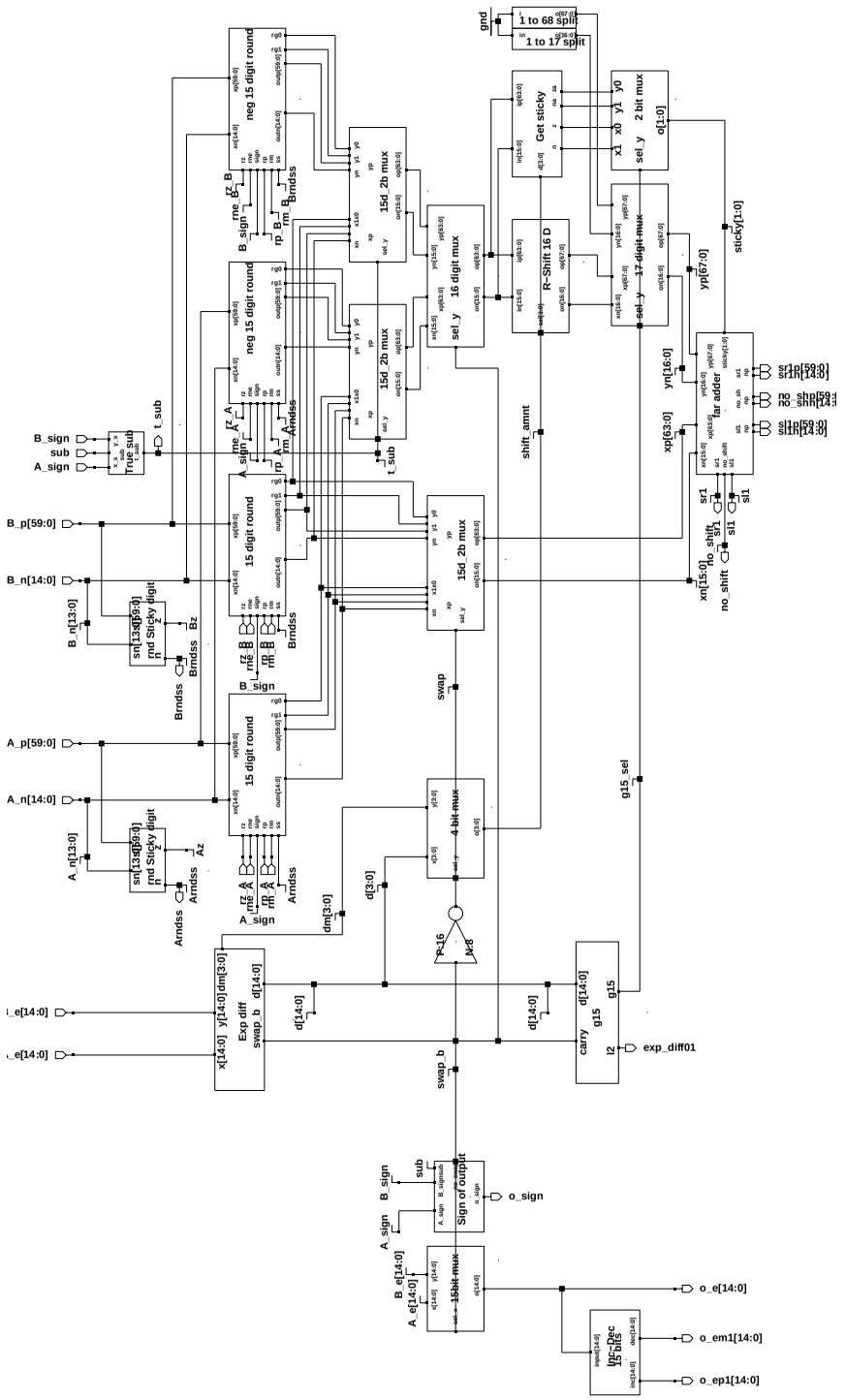


Figure C.9: The far path of the floating point adder.

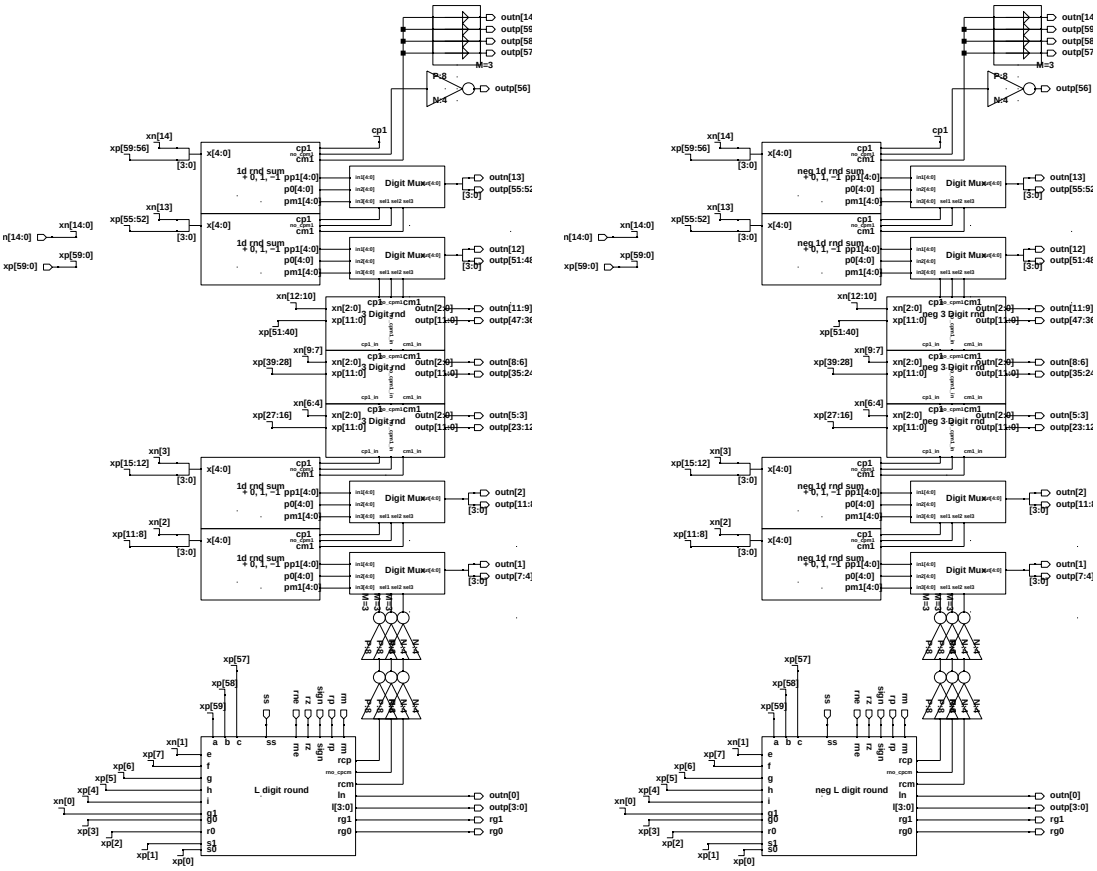


Figure C.10: Rounding blocks in the far path.

rules. If $carryout = 1$ then it is a positive difference ($A_e \geq B_e$) and if $carryout = 0$ then it means that $A_e < B_e$. The sign of the difference constitutes the swapping signal that decides which operand goes directly to the adder and which one is fed to the alignment shifter. The swapping signal also decides on the correct shifting amount to use (either that of $A_e - B_e$ or that of $B_e - A_e$).

The shifter is made wider than the operands in order to accommodate the guard and round digits. Any further digits that are shifted out are used to calculate the sticky digit. If the exponent difference is greater than 15 then all of the operand is shifted out and a multiplexer is used to select a zero value as the second input to the adder. For this, the result of the block checking if the exponent difference is larger than 15 is used. If the outputs of the exponent subtracter are labeled d_{14} (most significant bit) to d_0 (least significant) and the difference is positive ($carryout = 1$) then the condition is:

$$> 15_{pos} = d_{14} + d_{13} + \cdots + d_5 + d_4$$

If the difference is negative ($carryout = 0$) the condition to check should be done on the two's complement of the difference, or alternatively on the one's complement but checking for > 14 to account for the additional 1 of the two's complement. Hence the condition becomes:

$$> 15_{neg} = \overline{d}_{14} + \overline{d}_{13} + \cdots + \overline{d}_5 + \overline{d}_4 + (\overline{d}_3\overline{d}_2\overline{d}_1\overline{d}_0)$$

The complete block then generates the output named g_{15} which is given by $g_{15} = carryout(> 15_{pos}) + \overline{carryout}(> 15_{neg})$ as shown in Fig. C.11. In this same block, a check is made to see if the exponent difference is less than two (i.e. either zero or one). The logic at the bottom of Fig. C.11 is responsible for that.

The sticky digit is calculated from the outputs of the rounding blocks. For each digit at position i denoted by the bits $x_4x_3x_2x_1x_0$, we evaluate $\overline{z}_i = x_3 + x_2 + x_1 + x_0$ and $n_i = x_4$. These two values indicate if the digit is non-zero ($\overline{z}_i = 1$) and if it is negative ($n_i = 1$). According to the number system used, since a digit can never have the value of -16 then the case of $n_i = 1$ and $\overline{z}_i = 0$ is not possible. Since two digits are needed for G and R then the number of digits counted from the LSD that are used to determine the sticky value is equal to the exponent difference minus 2. For each of those digits, an enable signal, e_i, is set to 1. The enable is set to 0 for all the other digits. The boolean expressions giving the value of those enables are as shown in Table C.1. In the table, the digit designated as LSD_{-1} is the digit formed by the rounding logic in case the rounding is to the g_0 bit location while the digit designated by MSD_{+1} is the digit formed by the carry over from the MSD in the rounding blocks.

As a simplification of Table C.1, an alternative solution is to drop the g_{15} term from the equations and to proceed with the evaluation assuming $g_{15} = 0$. In parallel a sticky digit including all the digits of the number can be evaluated and then a multiplexer whose select line is g_{15} can be used to select the appropriate sticky. This is, in effect, the scheme used in this implementation as illustrated

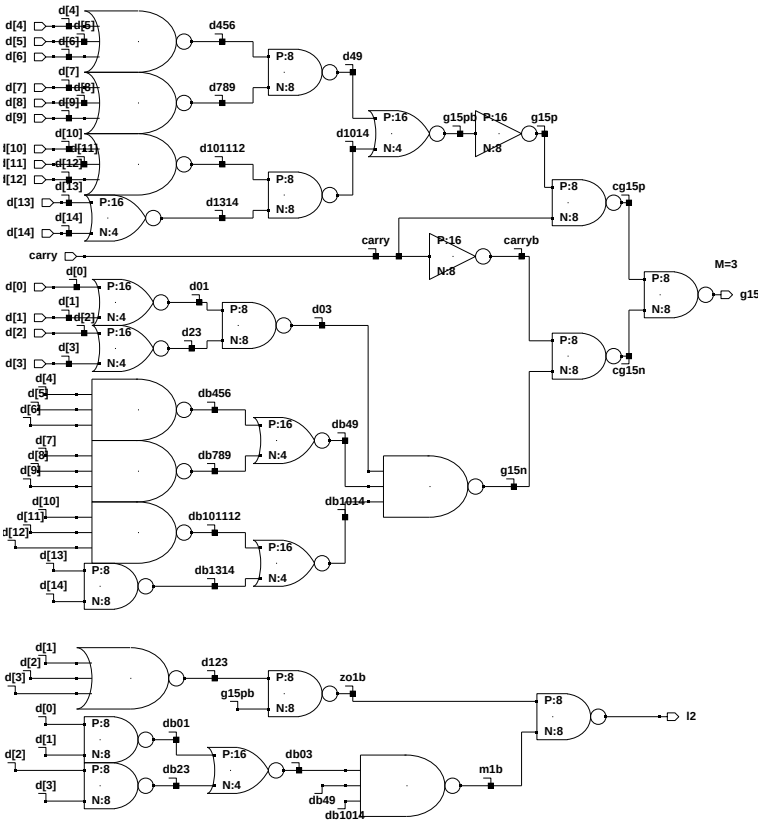


Figure C.11: Logic for checking the complete shift of the lesser operand. The lower part checks if the exponent difference is less than two.

Table C.1: Enable values for the Sticky digit calculation.

LSD_{-1}	$g_{15} + d_3 + d_2 + d_1$
LSD	$g_{15} + d_3 + d_2 + d_1 d_0$
	$g_{15} + d_3 + d_2$
	$g_{15} + d_3 + d_2(d_1 + d_0)$
	$g_{15} + d_3 + d_2 d_1$
	$g_{15} + d_3 + d_2 d_1 d_0$
	$g_{15} + d_3$
	$g_{15} + d_3(d_2 + d_1 + d_0)$
	$g_{15} + d_3(d_2 + d_1)$
	$g_{15} + d_3(d_2 + d_1 d_0)$
	$g_{15} + d_3 d_2$
	$g_{15} + d_3 d_2(d_1 + d_0)$
	$g_{15} + d_3 d_2 d_1$
	$g_{15} + d_3 d_2 d_1 d_0$
MSD	g_{15}
MSD_{+1}	g_{15}

by the multiplexer below the sticky generation block in Fig. C.9.

Thus the final values of $\overline{z}_i$ and n_i are given by:

$$\begin{aligned} \overline{z}_i &= (x_3 + x_2 + x_1 + x_0)e_i \\ n_i &= x_4 e_i \end{aligned}$$

Then these $\overline{z}_i$ and n_i are used to deduce two signals $\overline{z}$ and n indicating if the digits shifted out represent a non-zero value and whether it is negative or not. The value is non-zero if any of the digits (after the enable signal of course) is non-zero and it is negative if the MSD is negative or the MSD is zero and the second MSD is negative or both are zero and the third MSD is negative, ...

$$\begin{aligned} \overline{z} &= \overline{z}_0 + \overline{z}_{-1} + \overline{z}_{-2} + \overline{z}_{-3} + \cdots \\ n &= n_0 + z_0 n_{-1} + z_0 z_{-1} n_{-2} + z_0 z_{-1} z_{-2} n_{-3} + \cdots \end{aligned}$$

This is the same priority encoder that was presented in Fig. B.2.

The last major component of the far path is the adder shown in Fig. C.12 which provides three possible shifted versions of the output. The shifting for normalization in the far path is by at most one digit location to either the left or the right and is performed by re-wiring the signals. At the bottom of the figure the block labeled G, R, S evaluates three possibilities for the guard, round and sticky digits corresponding to the three possibilities for the normalization shift. That block is shown in Fig. C.13.

This adder in the far path needs to insure that at most one bit location shift is required in the rounding stage. The problem of shifting to a location more than one bit away arises if the MSD is equal to 1 and the following lower order digit is negative. If an N-recoding is used at the MSD location and then the resulting number checked for normalization that would solve the problem. With such a recoding the 1 is canceled if it is followed by a negative digit and the number is left shifted by one digit in the normalization. The block calculating the most significant digits in Fig. C.12 has two N-recoders for the two most significant digits as seen in Fig. C.14. The logic deciding whether or not a normalization shift is required and its direction appears in the top left corner of Fig. C.12. This logic operates on the outputs of the adder after the N-recoding.

C.3 Conclusions

Learning about the use of the design tools and starting the design implementation took less than a month. Completing this large design (125 238 transistors) and correcting all the mistakes that were discovered along the way took much longer. Slightly over six months were needed to finish all the implementation details of the adder. That includes the time for insuring the logical correctness and the speed simulations. Building small modules and reusing those modules in larger structures

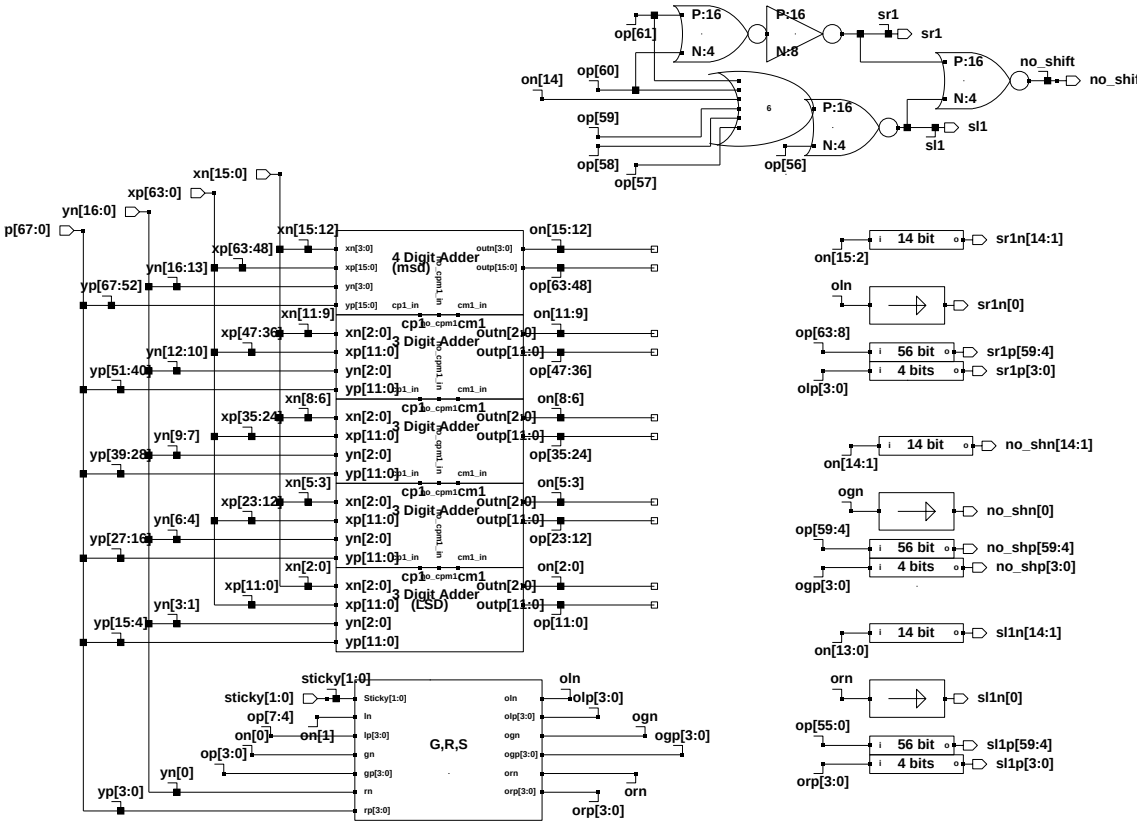


Figure C.12: Adder of the far path.

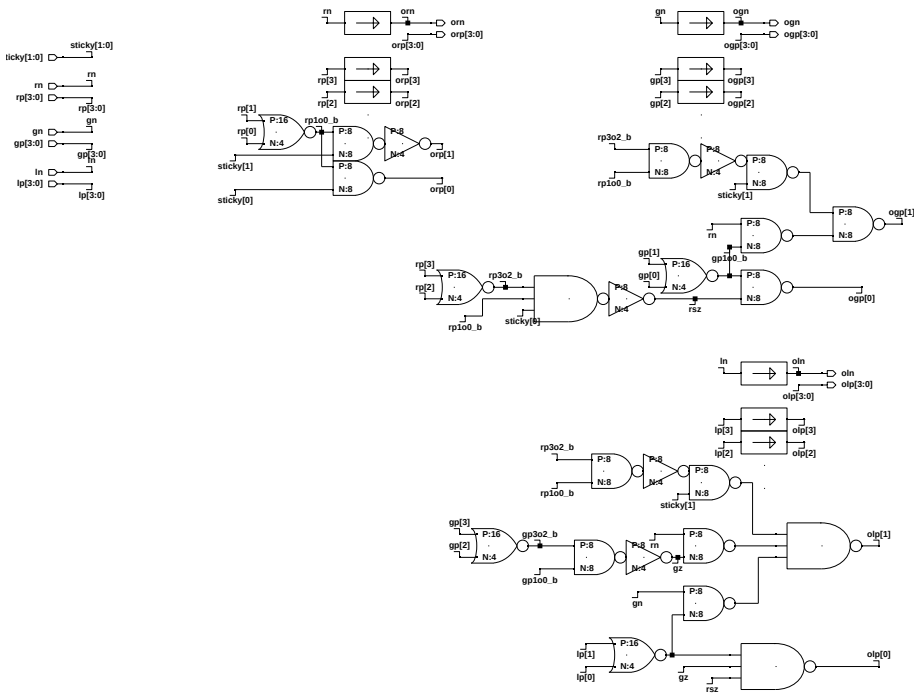


Figure C.13: Evaluation of the guard, round and sticky digits in the far path.

helped greatly in speeding the process. The cancellation path was done in just over a month because it reuses a large number of the smaller components from the far path which was finished first. The final outcome is an adder faster than conventional designs and capable of performing all the IEEE rounding modes.

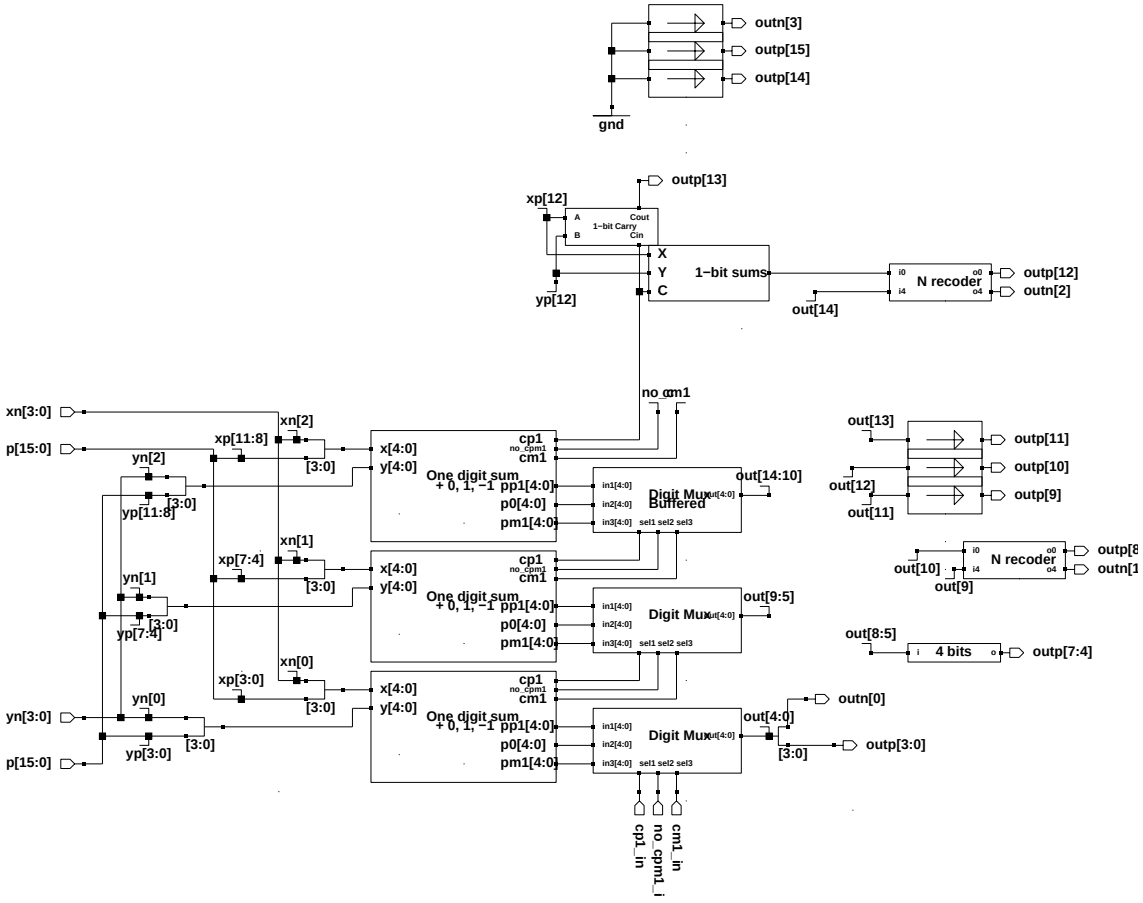


Figure C.14: N-recoding in the most significant part of the far path adder.

Appendix D

Implementation details of the multiplier

Similar to the floating point adder, the proposed design for the multiplier with the four IEEE rounding modes is implemented at the transistor level using a scalable CMOS technology with $n = 53$, $f = 3$ and $r = 4$. The schematic entry tool *sue* is used to describe the circuit connections and the design is simulated for functionality at the logic level using *verilog* and for speed at the transistor level using the switch level simulator *irsim*.

Fig. D.1 shows the schematic diagram of the floating point multiplier. The *recode* Y block to the left does the Booth recoding, chops Y at the rounding location and provides the information necessary for correctly rounding Y to the partial product tree (PPT) block to its right. The block labeled *Prepare* X at the top chops X, produces mX and the information required for correctly rounding X. After the PPT, a $[4:2]$ compressor is used and then the final adder.

D.1 Basic blocks

Two important basic blocks repeatedly used are the $[4:2]$ compressor and the partial product multiplexer. The design of the $[4:2]$ compressor is shown in Fig. D.2. As the parametric model presented earlier predicts this unit takes approximately 3 *FO4* gate delays. The compressor is used in the form of a row with possibly simple half adders or full adders to sum the partial products. The need for half adders or full adders arises at the edges due to the relative shift between the partial products. An example with a 58 bit wide row of compressors is shown in Fig. D.3.

This row is the one used to sum the positive vectors of four consecutive partial products as illustrated by Fig. D.4. On the other hand, the negative vectors are sparse vectors and can be combined together by wiring the bits in the correct location without any hardware to sum them as

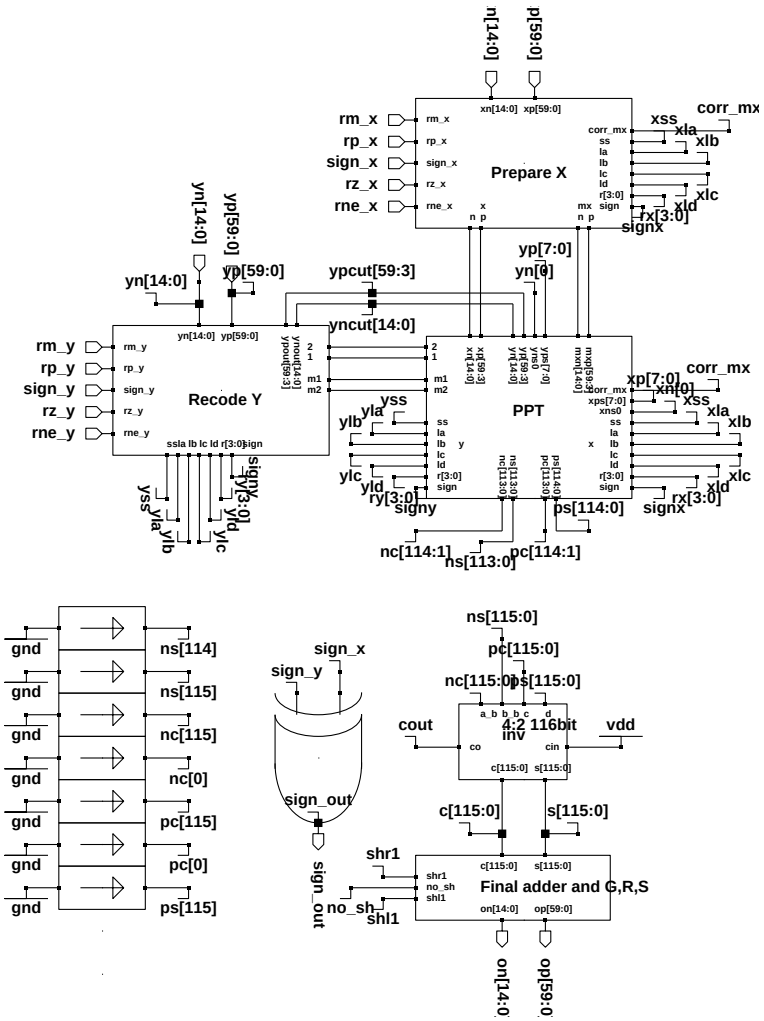


Figure D.1: Schematic of the floating point multiplier.

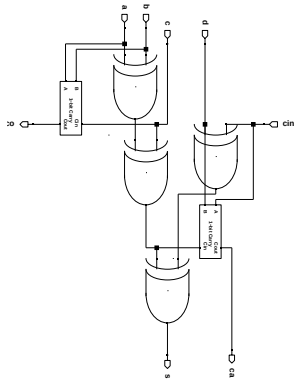


Figure D.2: The [4 : 2] compressor.

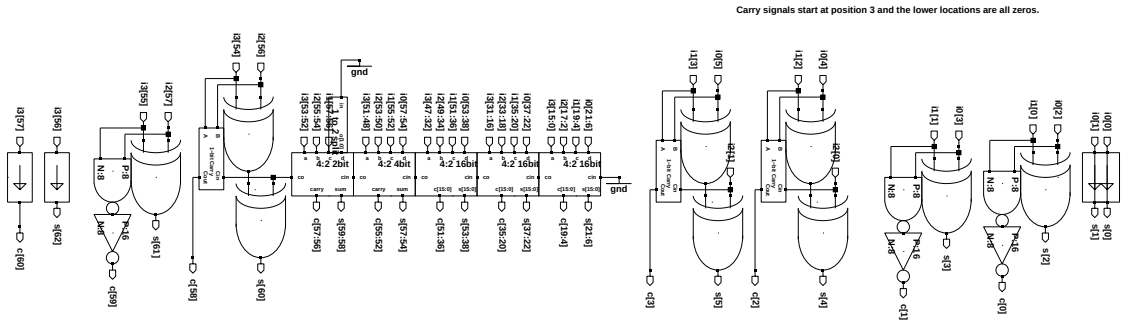


Figure D.3: A 58 bits row of $[4:2]$ compressors.

seen at the bottom right of the figure. This Fig. D.4 actually gives out the result of multiplying the chopped X by two digits of Y recoded as the signals $two[3:0]$, $one[3:0]$, $m-one[3:0]$ and $m-two[3:0]$ coming from the left direction.

The block labeled *PPmux 15D* in Fig. D.4 is an array of the basic partial product multiplexer shown in Fig. D.5. Depending on the selection signals coming from the recoded Y, the multiplexers choose either X or mX or their shifted versions. The individual multiplexers used are a simple implementation of the boolean relation $out = A\,sel_A + B\,sel_B + C\,sel_C + D\,sel_D$ as shown in Fig. D.6. If all the selectors are low the output is low indicating a value of zero. With the use of these multiplexers, the Booth recoding of Y can be made simpler as indicated in Fig. 5.2 where there is no need to generate special signals for ± 0.

D.2 Details of the floating point multiplier

As mentioned earlier, in Fig. D.1, the block labeled *prepare X* is responsible for generating mX. Two minor properties must be noted:

1. $(-X)_{chopped} \neq -(X_{chopped})$. For example, let the LSD of X be $0\,0110 = (6)$ and let it be chopped to become $0\,0100 = (4)$. The negative of the original LSD is $1\,1010 = (-6)$ and if it is chopped at the same location the result is $1\,1000 = (-8)$. Obviously this is not representative of the chopped X. Hence, there is a need to decide on the cutting location first and then negate the number.

2. If by chopping X a digit equal to $1\,0000 = (-16)$ is produced then the corresponding negated digit is $+16$ but the simple rules of negation produces $1\,0000 = (-16)$. The solution implemented here is to give an mX LSD of all zeros in this special case and then later apply a special correction for every time mX is used.

The details of the *prepare X* block are shown in Fig. D.7. The top right determines the leading one in the MSD, the top middle produces the signed sticky for X to correctly round it using the

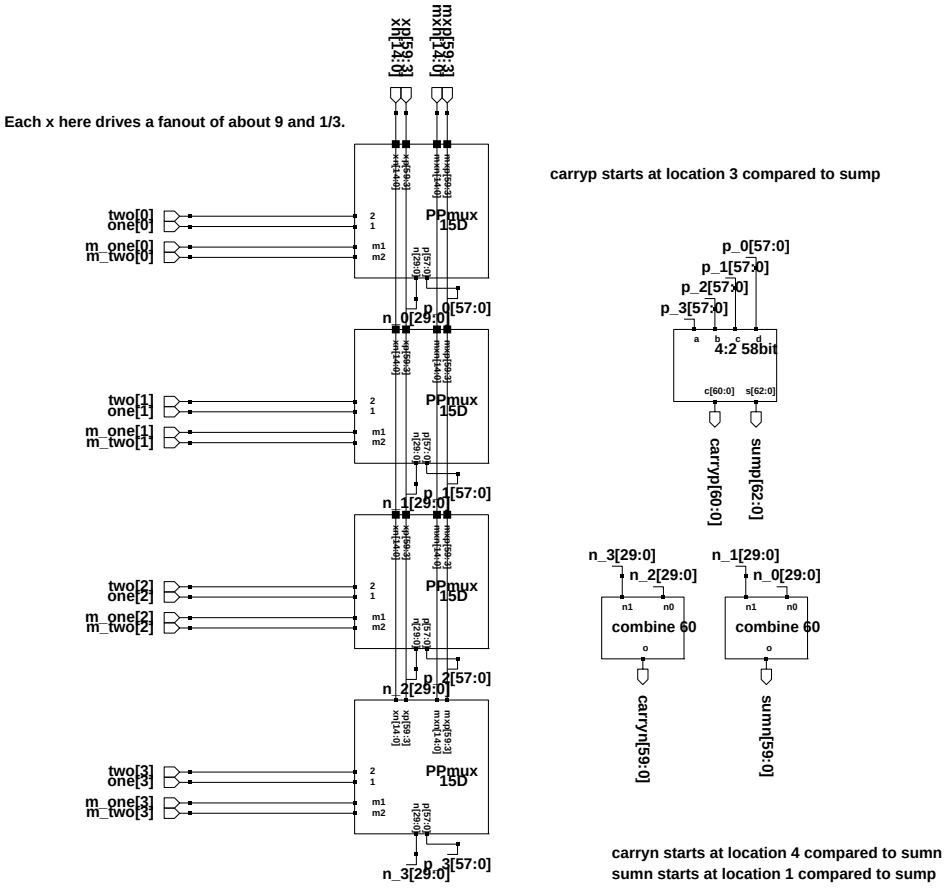


Figure D.4: Generation and compression of 4 partial products.

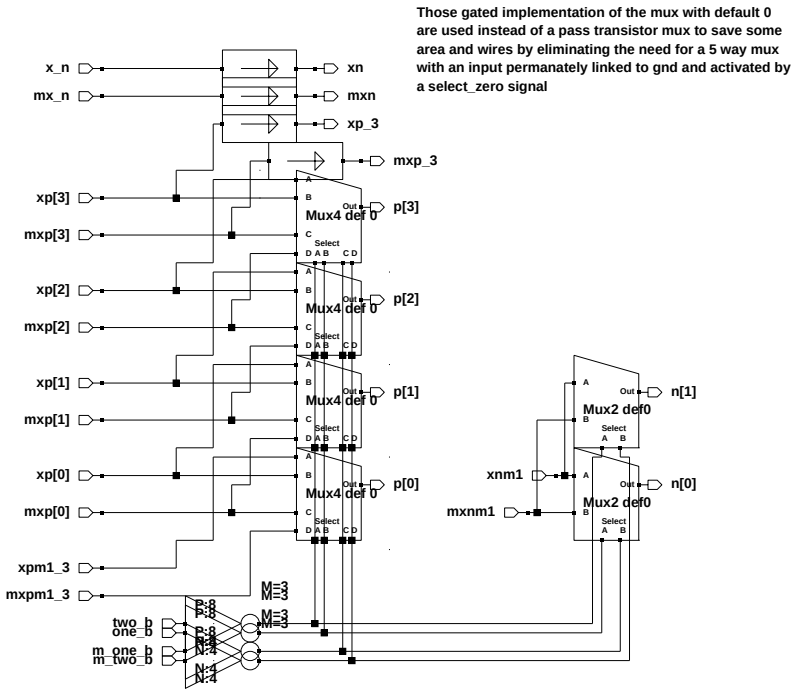


Figure D.5: Partial product multiplexer.

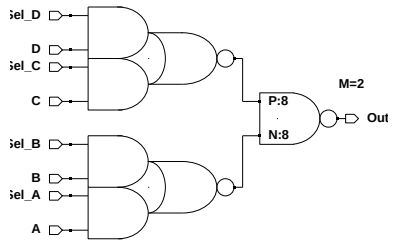


Figure D.6: The four to one multiplexer with default equal to zero.

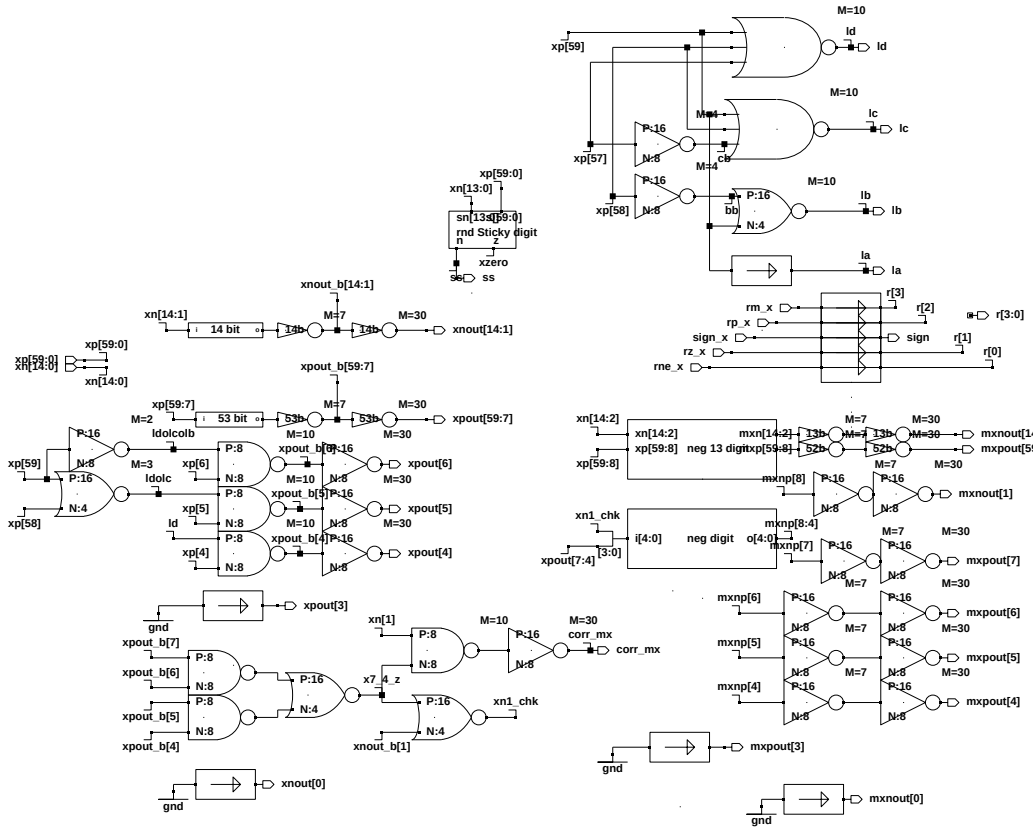


Figure D.7: The preparation of mX.

signals buffered at the middle right. The bottom left is for chopping X and generating near the middle of the figure a signal labeled *corr-mx* to check for the special case of a chopped LSD equal to -16. The bottom right side is for negating X and buffering mX.

The signal $corr - mx$ is used in the block shown in fig. D.8 to enable a compensation constant for each time the Booth recoder produced a minus one or minus two output. Basically, for the case of $m - one$, a sparse vector is formed with a bit equal to one if $corr - mx = 1$ and the corresponding $m - one$ bit from the Booth recoder is also one. Similarly for the case of $m - two$ but the second sparse vector is one bit shifted relative to the first sparse vector. They are both then combined by wiring them to the correct bits of the output.

The correction for the use of mX is then delivered to a unit handling the addition of the two rounding corrections from $(b_x + r_x)Y_{chopped}$ and $(b_y + r_y)X_{chopped}$ to one of the partial products as presented in Fig. D.9 where the bottom left two blocks represent the circuit described in Fig. B.3.

The unit in Fig. D.9 is then combined with that of Fig. D.4 to form the top module to the left of Fig. D.10. The module labeled *corr-rxry* in the top middle of the figure is the one already presented

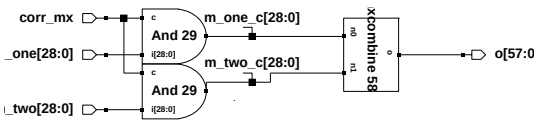


Figure D.8: Correction for the use of mX in case of an LSD of -16.

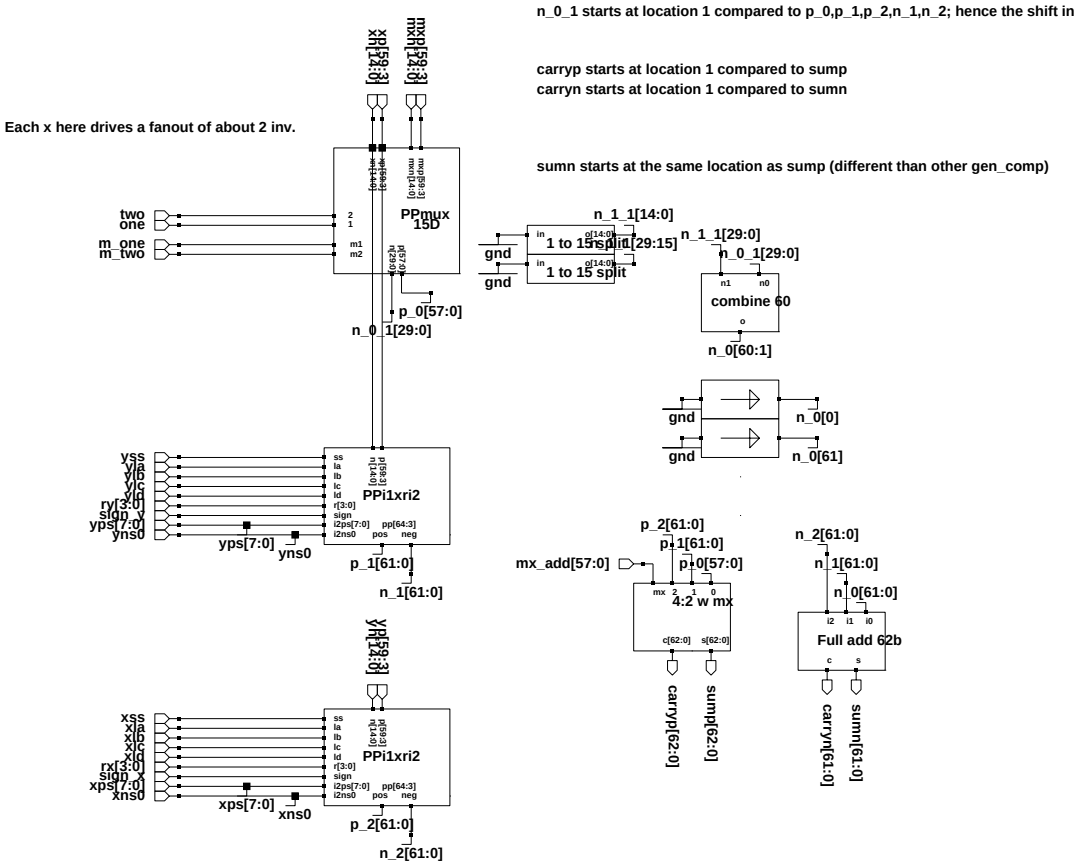


Figure D.9: Inclusion of the correction for mX and the two special rounding PPs.

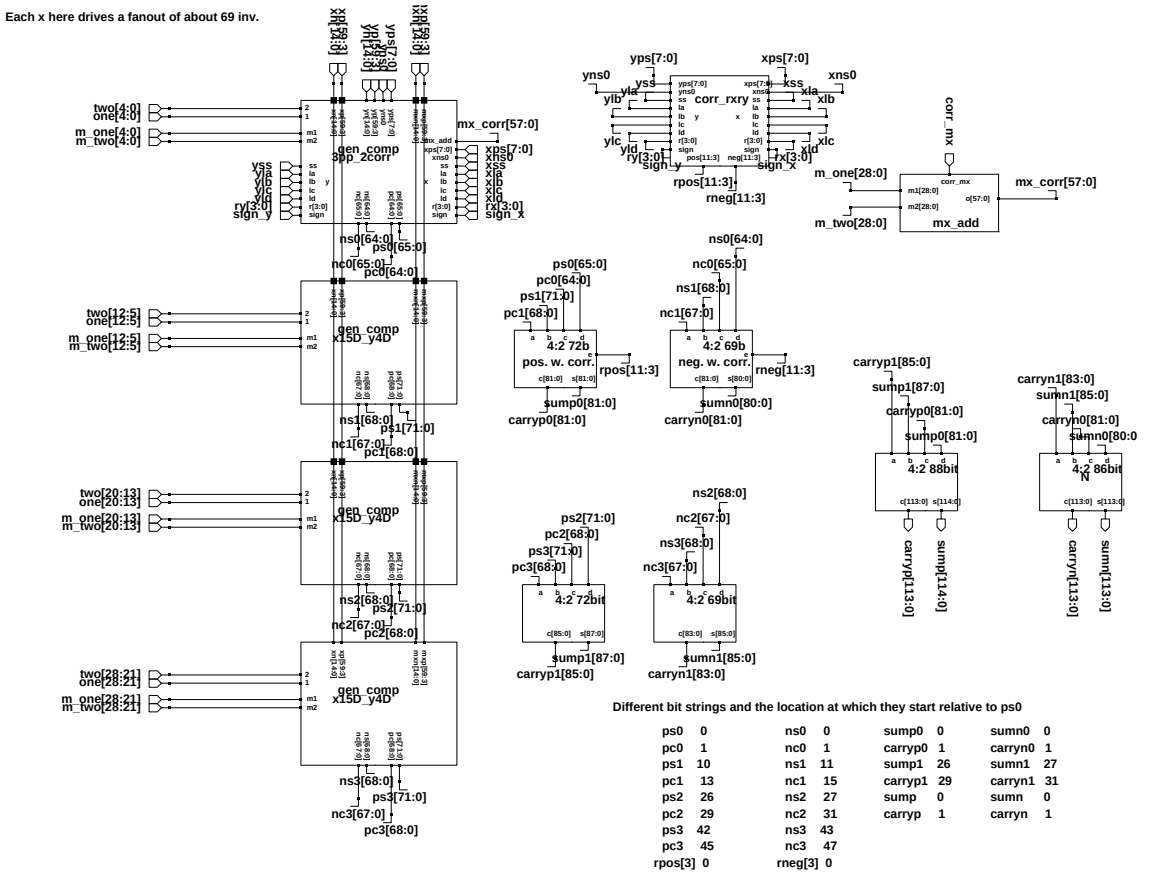


Figure D.10: The complete partial product tree.

in Fig. B.4 to produce the correction for $(b_x + r_x)(b_y + r_y)$. This correction does not involve any high significance digits and hence it is guaranteed to be at the lower significant portion of the final result of the multiplier. On the other hand, any of the $[4:2]$ compressors has an empty range of bits at the lower significance due to the shifting of the higher order partial products. The positive and negative outputs of that correction block are thus easily included in the positive and negative compressors directly below it as shown in the middle of Fig. D.10. The relative shift between the different partial products is noted in the bottom right part of the figure. These shifts are reported relative to the first partial product which has the lowest significant bits.

Due to the relative shift, the outputs of the partial product tree module are padded with zeros as was shown to the lower left part of Fig. D.1 before summing them in the inverting $[4:2]$ compressors row. That row of compressors complements the vectors corresponding to the negative sum (nc and ns in the figure) and adds them to the vectors of the positive sum (pc and ps). An extra one is added at the least significant side as the carry into the row as shown in Fig.D.1. However, to correctly provide the two's complement of the two negative vectors an additional one should be added for

each of them. With the current implementation, the carry and sum bit vectors resulting from the compression are equal to $pc + ps - nc - ns - 1$. This negative one is intentionally left to facilitate the calculation of the sticky digit.

The sticky digit calculation in multipliers is done in a number of ways: [77, 73]

1. The simplest conceptually is to implement a complete carry propagation adder for the lower half of the result and then use a logic tree of gates to form a large *OR* gate for all the bits of that lower half. Each of those two steps is an $\mathcal{O}(\log n)$ operations and hence if they are done in sequence the circuit is quite slow.

2. The sticky digit can be evaluated directly from the inputs by counting the number of trailing zeros in each operand and adding them to find the number of trailing zeros in their product. This method is correct for any representation in which the base β is a prime number that cannot be factored. Our case here has a non-prime base $\beta = 16$. If, for example, the least significant non-zero digits in the operands are 2 and 8 then an additional zero is generated. The number of trailing zeros in the product is larger than the sum of the number of trailing zeros in both operands.

3. Another method to find the sticky digit from the partial product bits exists. It depends on having all of the partial products as positive numbers and not using the Booth algorithm. In that method, the logical *OR* of all the bits of the partial product array below the rounding bit is evaluated to give the sticky bit. This method works when all the partial products are positive because the first column in the partial product array yielding a non-zero result is proved to contain at most a single 1 [77]. Since in our floating point multiplier Booth recoding is used, this method cannot be implemented.

4. An improvement on the previous method to allow the use of Booth recoding was suggested by Bewick [73]. Basically, we want to know if the result of adding the sum and carry bits of the lower part of the product is exactly equal to zero or not. If the two bit vectors constituting the part of the sum and carry being checked are called A and B then we want to check if $A + B = 0$. Alternatively, we can check for $A + B - 1 = -1 = (111 \cdots 11111)_{\beta=2}$. If a constant equal to -1 is injected somehow in the partial product compression, we get at the end two bit vectors to which we need to add $+1$ to get the correct result. However, getting the sticky is now equivalent to evaluating the logical *XOR* at each bit position of the two vectors followed by calculating the logical *NAND* of all those *XOR* outputs. This last method is in fact a special case of checking for $A + B$ being equal or not to some constant K [83, 84] and can be used with some modifications in our design.

Only the first method of evaluating the sticky digit requires the actual computation of the sum of the lower portion of the product. In the last three methods, the hardware doing that computation

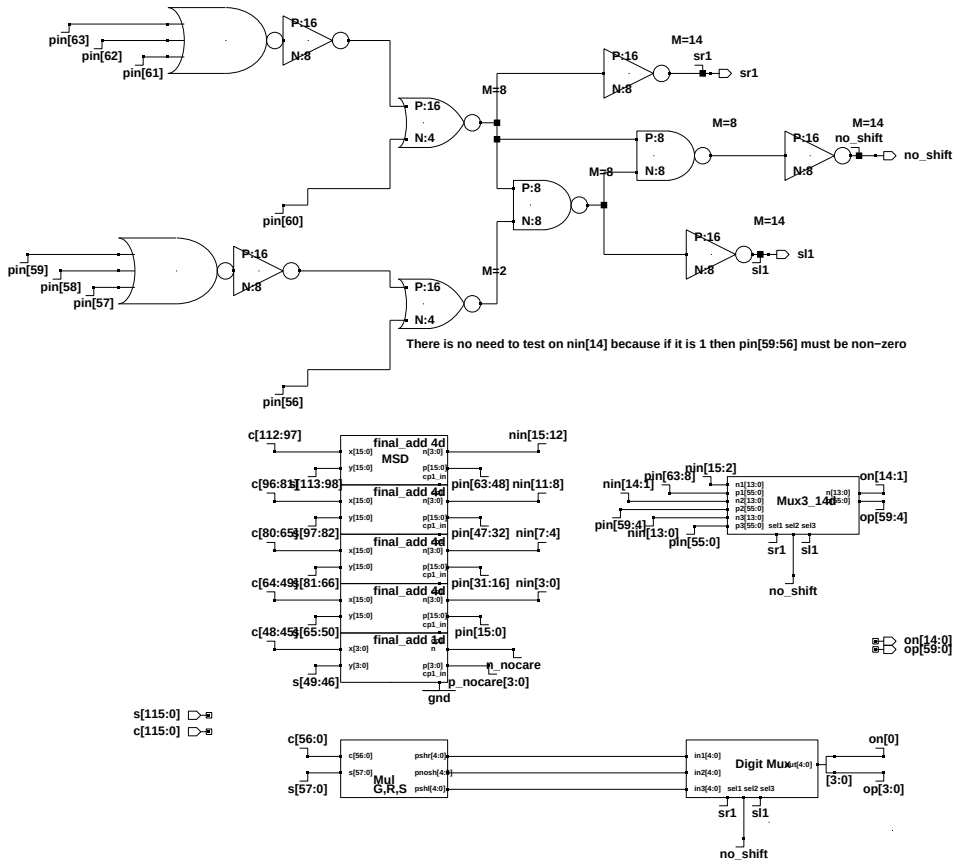


Figure D.11: The final adder of the multiplier.

can be eliminated. The only required hardware is the gates needed for generating the sticky digit as well as any potential carry into the higher remaining part from the truncated lower part. But, before explaining the implementation of the sticky digit, let us first introduce the details of the final adder since a good understanding of the normalization issues is necessary to explain the sticky digit calculation.

Fig. D.11 shows the final adder of the multiplier. The bottom part is for the evaluation of the sticky digit while the middle part is the SD adder and the top is the signals checking for normalization.

Each digit of the SD adder is implemented as shown in Fig. D.12. The simplicity of this circuit should be contrasted to the corresponding circuit in the floating point adder shown in Fig. C.1. This simplicity is the reason for the lower estimation for the time delay of this adder as described in section 5.3.

The normalization logic at the top of Fig. D.11 depends on the following Lemma.

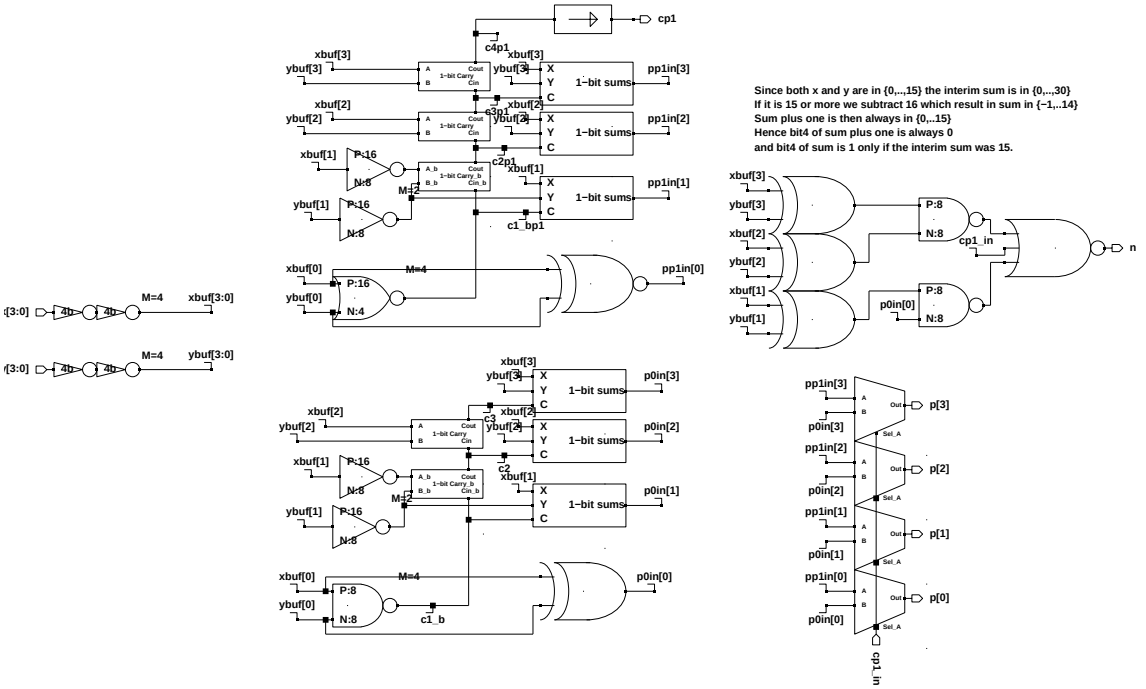


Figure D.12: One digit of the final adder of the multiplier.

Lemma D.2.1 *The normalization in the floating point multiplier can be by at most one digit to the left or one digit to the right.*

Proof: The operands X and Y of the multiplier are normalized numbers and the minimum value of the MSD is thus 1. The condition required for the rounding logic to shift by only one bit location insures that the operands cannot be of the form $1\,(-ve)\cdots$. However the operands can have the MSD equal to 1 and the rest of the significand being a negative number as in the form of $1\,0\,(-ve)\cdots$. On the other hand, the maximum value of the MSD is 15. Hence, the value of the operands lies in the range $[1-\epsilon, 16[$ where ϵ is a small positive number less than 16^{-1}. The product of two such values is in $[1-\delta, 256[$ where δ is derived from multiplying the least possible operands.

$$(1-\epsilon)(1-\epsilon) = 1 - 2\epsilon + \epsilon^2$$

So, $\delta < 2 \times 16^{-1}$ and the minimum value of $1-\delta$ is then larger than $1 - 2 \times 16^{-1}$.

If the value of the product is in $[16, 256[$ one digit right shift reduces it to the required range for normalized numbers. If the value of the product is in $[1-2\times16^{-1}, 1[$ one digit left shift is enough to normalize it. Otherwise, the value is in $[1, 16[$ and the product is already normalized with no need to shift it. $\diamond$

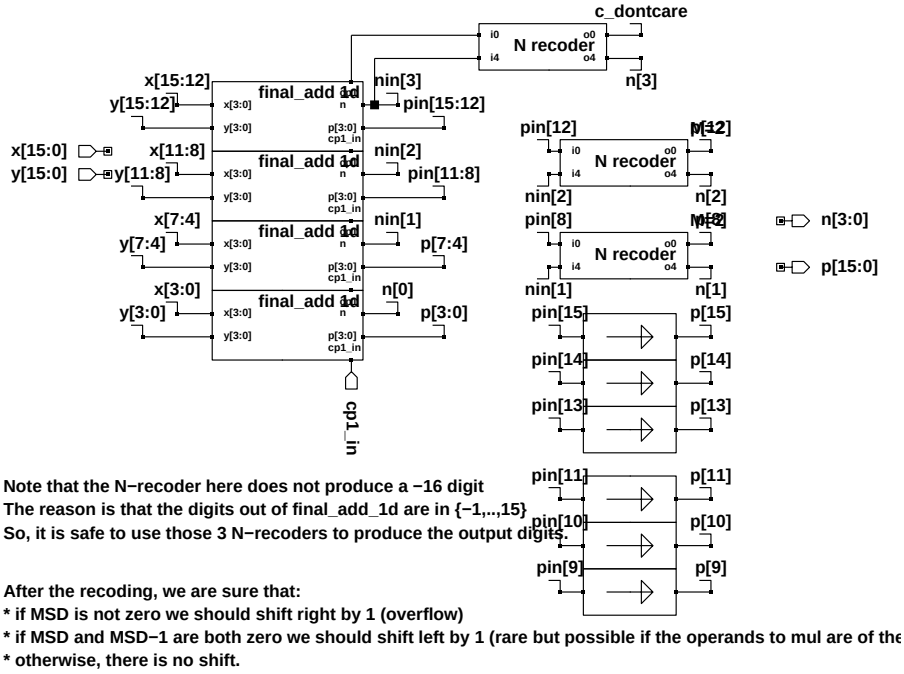


Figure D.13: The N-recoding in the most significant digits of the final adder.

In the proof, the value of the product is equivalent to the value of the non-redundant representation. However, the SD adder produces a redundant representation. It is that representation that is checked. Also, if this representation has an MSD of 1 and a following digit that is negative an N-recoding must be used to insure the property of normalized numbers needed for the rounding as shown in Fig. D.13 representing the most significant part of the adder of Fig. D.11. The multiplexers to the right of Fig. D.11 use the shifting signals to select the correct output for the multiplier.

In calculating the sticky digit, note that the lower part of the product arrives at the final adder stage before the higher part of the product. This fact has been documented [60, 76] and exploited to implement fast yet simple final adders for conventional multipliers. That fact is also exploited in this implementation. Since the last $[4:2]$ compression stage before the final adder introduced a negative one into the product, an extra positive one must be added to the lower part to compensate it. Because of the early arrival of the lower parts of the sum and carry bit vectors, a simple ripple carry chain is enough to get the carry out of adding the lower parts of both vectors as well as the extra positive one. In fact three carries are generated from this ripple carry chain corresponding to the three possibilities for normalization as shown at the bottom of Fig. D.14. The left part of the figure is a tree of *XOR* gates followed by a logical *NAND* to evaluate the sticky bit as explained above. Three stickies corresponding to the three possibilities of normalization are also calculated.

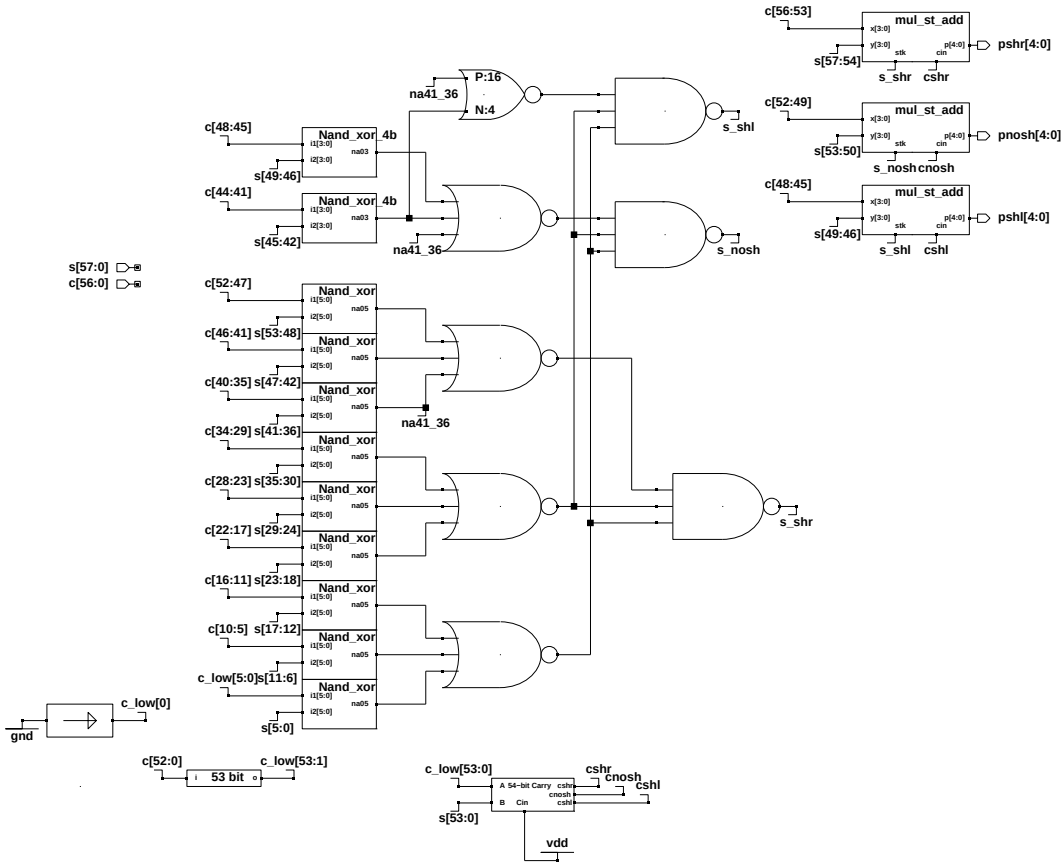


Figure D.14: Evaluation of the guard, round and sticky digits.

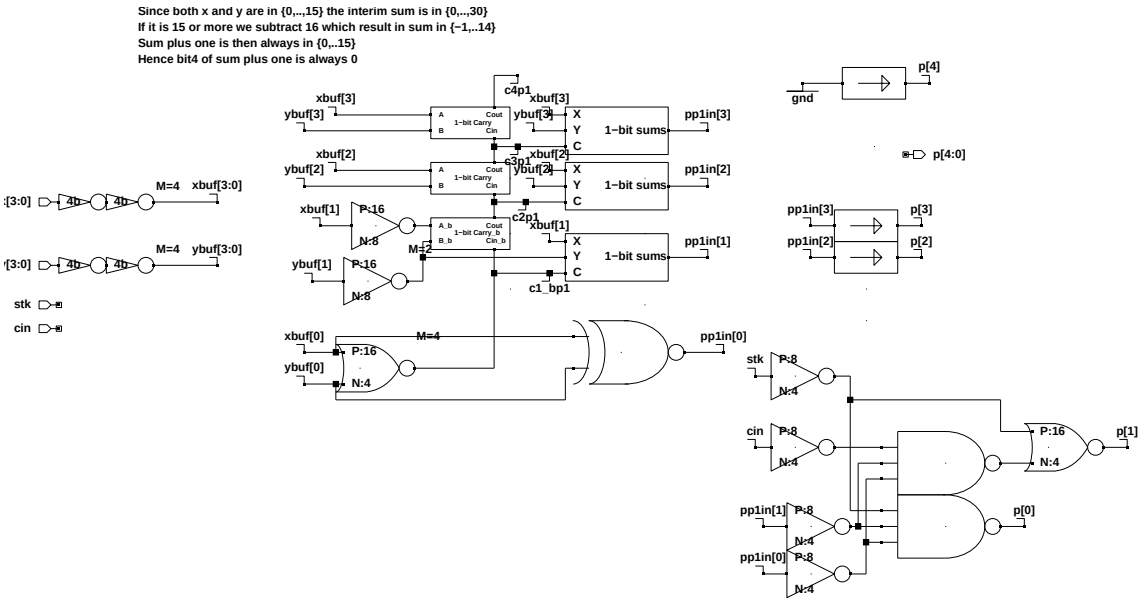


Figure D.15: The special adder for the sticky digit calculation.

The three possible carries and the three possible stickies are forwarded to the blocks at the top right of Fig. D.14 to evaluate, for the three possibilities of normalization, the digit representing the guard, round and sticky digits. Each of those blocks is implemented as shown in Fig. D.15. It is a special adder that adds the sum and carry bits corresponding to the digit directly below the LSD of the final product. In effect, it adds them assuming a carry into this digit of +1 then uses the actual calculated carry and sticky bit representing all the bits below this digit to adjust the result. In the figure, the preliminary result of the sum of the inputs plus one is labeled $pp1in$ and the actual result is $p[4:0]$. Referring back to the nomenclature of the rounding logic (section 4.2.2), $p[4] = g_1$, $p[3] = g_0$, $p[2] = r$, $p[1] = s_1$ and $p[0] = s_0$. The adjustment occurs only in the two least significant bits corresponding to s_1 and s_0 as seen from the lower right portion of the figure. Simply, the resulting sticky is equal to 0 ($s_1 s_0 = 00$) if the incoming sticky from the lower bits is zero and both of the least significant bits of $pp1in$ are also zero. If the incoming sticky is 0 that means that the logical XOR of all the lower bits evaluates to 1 and the incoming carry must be equal to 1 (remember the extra +1 added at the least significant bit.) The $pp1in$ is calculated taking into account a carry into the digit equal to 1 so the upper bits of $pp1in$ are correct and represent g_1, g_0 and r.

If the incoming sticky is zero but either $pp1in[0]$ or $pp1in[1]$ is not zero then the resulting sticky $s_1 s_0 = 01$. Once again, the incoming carry must be one and the values of $pp1in[4:2]$ correctly represent g_1, g_0 and r.

If the incoming sticky is 1 then the extra +1 added at the least significant bit is not propagated all the way. A sticky of 1 means that at least one of the *XOR* gates produces a zero result. The propagation of the added +1 is stopped at this position where the result of the *XOR* is zero. The other positions higher than this one may or may not generate a carry. Hence, the incoming carry can be either 0 or 1. Remember that $pp1in$ is calculated assuming a carry of one. So, if both the sticky and the carry are equal to 1, $s_1 s_0$ is set to 01 and the upper bits $pp1in[4:2]$ correctly represent g_1, g_0 and r.

On the other hand, if the sticky is 1 and the carry is 0, a more detailed analysis of the possible combinations for $pp1in[1:0]$ is needed while keeping in mind that $pp1in$ is calculated assuming a carry of 1. This means that we need to compensate for that assumed carry as in the following table:

Calculated		Compensated		Output	
$pp1in[1]$	$pp1in[0]$	$pp1in[1]$	$pp1in[0]$	s_1	s_0
1	1	1	0	0	1
1	0	0	1	0	1
0	1	0	0	0	1
0	0	0	−1	1	1

The first three lines are easily compensated and since the incoming sticky is one the resulting $s_1 s_0$ are given as 01. The last line is where a simple trick is used. To compensate the assumed carry we introduce a negative bit at this location that combines with the incoming sticky to give $s_1 s_0 = 11$, i.e. a resulting sticky digit of -1. This trick shows the elegance of working with redundant digits, you can apply a late correction for an early assumption without the need to redo the computation: the bits $pp1in[4:2]$ once more correctly represent g_1, g_0 and r.

This long derivation of the sticky digit leads at the end to the simple logic gate implementation shown at the bottom right of Fig. D.15.

D.3 Conclusions

A three months period was needed to complete the full design of the multiplier. This is half the design time of the adder despite the fact that the muliplier is larger (209 732 transistors versus about 125 238 transistors for the adder.) The reuse of a number of parts from the adder as well as the repetitive structure of the [4 : 2] compressor rows helped to cut down the design time. Those three months include the period used for checking the correctness of the Verilog model as described in section 5.4 as well as the timing simulations.

Bibliography

[1] A. A. Liddicoat, *High-Performance Arithmetic for Division and The Elementary Functions*. PhD thesis, Stanford University, Feb. 2002.

[2] A. A. Liddicoat and M. J. Flynn, "High-performance floating point divide," in *Proceedings of the Euromicro Symposium on Digital System Design*, pp. 354–361, Sept. 2001.

[3] M. ibn Musa Al-Khawarizmi مُحَمَّدٌ بنُ مُوسَى الخَوَارِزمِيِّ, *The keys of Knowledge,* مَفَاتِيحُ العُلُومِ. around 830 C.E.

[4] "IEEE standard for binary floating-point arithmetic," Aug. 1985. (ANSI/IEEE Std 754-1985).

[5] "IEEE standard for radix-independent floating-point arithmetic," Oct. 1987. (ANSI/IEEE Std 854-1987).

[6] W. Kahan, "Digit-set conversions: Generalizations and applications," May 1996. `http://www.cs.berkeley.edu/~wkahan/ieee754status/ieee754.ps`.

[7] C. Severance, "IEEE 754: An interview with William Kahan," *IEEE Computer magazine*, vol. 31, pp. 114–115, Mar. 1998.

[8] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 3 ed., 2003.

[9] S. F. Oberman, *Design Issues in High Performance Floating Point Arithmetic Units*. PhD thesis, Stanford University, Nov. 1996.

[10] O. J. Morales, "An SBNR floating-point convention," in *IEEE Region 5 Conference, 1988: Spanning the Peaks of Electrotechnology*, pp. 6–10, Mar. 1988.

[11] D. W. Matula and A. M. Nielsen, "Pipelined packet-forwarding floating point: I. foundations and a rounder," in *Proceedings of the 13th IEEE Sympsoium on Computer Arithmetic, Asilomar, California, USA*, pp. 140–147, July 1997.

[12] S. F. Oberman, "Floating point division and square root algorithms and implementation in the AMD-K7^{TM} microprocessor," in *Proceedings of the 14th IEEE Sympsoium on Computer Arithmetic, Adelaide, Australia*, pp. 106–115, Apr. 1999.

[13] E. M. Schwarz, R. M. Smith, and C. A. Krygowski, "The S/390 G5 floating point unit supporting hex and binary architectures," in *Proceedings of the 14th IEEE Sympsoium on Computer Arithmetic, Adelaide, Australia*, pp. 258–265, Apr. 1999.

[14] S. Winograd, "On the time required to perform addition," *Journal of the Association for Computing Machinery*, vol. 12, pp. 277–285, Apr. 1965.

[15] S. Winograd, "On the time required to perform multiplication," *Journal of the Association for Computing Machinery*, vol. 14, pp. 793–802, Oct. 1967.

[16] P. M. Spira, "Computation times of arithmetic and boolean functions in (d, r) circuits," *IEEE Transactions on Computers*, vol. C-22, pp. 552–555, June 1973.

[17] S. Waser and M. J. Flynn, *Introduction to Arithmetic for Digital Systems Designers*. Holt, Rinehard & Winston, New York, 1982.

[18] B. Rejeb, H. Henkelmann, and W. Anheier, "Integer division in residue number system," in *The 8th IEEE International Conference on Electronics, Circuits and Systems*, vol. 1, pp. 259–262, Sept. 2001.

[19] W. L. Freking and K. K. Parhi, "Modular multiplication in the residue number system with application to massively-parallel public-key cryptography systems," in *Thirty-Fourth Asilomar Conference on Signals, Systems, and Computers, Asilomar, California, USA*, vol. 2, pp. 1339–1343, Oct. 2000.

[20] M. N. Mahesh and M. Mehendale, "Low power realization of residue number system based FIR filters," in *Thirteenth International Conference on VLSI Design, Calcuta, India*, pp. 30–33, Jan. 2000.

[21] P. Fernández, J. Ramírez, A. García, L. Parrilla, and A. Lloris, "A new rns architecture for the computation of the scaled 2d-dct on field-programmable logic," in *Thirty-Fourth Asilomar Conference on Signals, Systems, and Computers, Asilomar, California, USA*, vol. 1, pp. 379–383, Oct. 2000.

[22] B. Parhami, "RNS representations with redundant residues," in *Thirty-Fifth Asilomar Conference on Signals, Systems, and Computers, Asilomar, California, USA*, vol. 2, pp. 1651–1655, Nov. 2001.

[23] T. Stouraitis and C. Chen, "Hybrid signed digit logarithmic number system processor," *IEE Proceedings-E*, vol. 140, pp. 205–210, July 1993.

[24] M. J. Flynn and S. f. Oberman, *Advanced Computer Arithmetic Design*. John Wiley & Sons, Inc., 2001.

[25] A. Avizienis, "Signed-digit number representations for fast parallel arithmetic," *IRE Transactions on Electronic Computers*, vol. EC-10, pp. 389–400, Sept. 1961.

[26] B. Parhami, "Generalized signed-digit number systems: A unifying framework for redundant number representations," *IEEE Transactions on Computers*, vol. 39, pp. 89–98, Jan. 1990.

[27] B. Parhami, *Computer Arithmetic Algorithms And Hardware Designs*. http://www.oup-usa.org: Oxford University Press, 2000.

[28] I. Koren, *Computer Arithmetic Algorithms*. Englewood Cliffs, N.J.: Prentice Hall, 1993.

[29] P. Kornerup, "Digit-set conversions: Generalizations and applications," *IEEE Transactions on Computers*, vol. 43, pp. 622–629, May 1994.

[30] G. M. Blair, "The equivalence of twos-complement addition and the conversion of redudnat-binary to twos-complement numbers," *IEEE Transactions on Circuits and Systems-I*, vol. 45, pp. 669–671, June 1998.

[31] B. Parhami, "On the implementation of arithmetic support functions for generalized signed-digit number systems," *IEEE Transactions on Computers*, vol. 42, pp. 379–384, Mar. 1993.

[32] D. S. Phatak and I. Koren, "Hybrid signed-digit number systems: A unified framework for redundant number representations with bounded carry propagation chains," *IEEE Transactions on Computers*, vol. 43, pp. 880–891, Aug. 1994.

[33] M. I. Furgeson and M. D. Ercegovac, "A multiplier with redundant operands," in *Thirty-Third Asilomar Conference on Signals, Systems, and Computers, Asilomar, California, USA*, vol. 2, pp. 1322–1326, Oct. 1999.

[34] H. Makino, Y. Nakase, and H. Shinohara, "A 8.8-ns 54×54-bit multiplier using new redundant binary architecture," in *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 202–205, Oct. 1993.

[35] N. Takagi, H. Yasuura, and S. Yajima, "High-speed VLSI multiplication algorithm with a redundant binary addition tree," *IEEE Transactions on Computers*, vol. c-34, pp. 789–796, Sept. 1985.

[36] M. D. Ercegovac and T. Lang, "Fast multiplication without carry-propagate addition," *IEEE Transactions on Computers*, vol. 39, pp. 1385–1390, Nov. 1990.

[37] J.-J. J. Lue and D. S. Phatak, "Area × delay ($a.t$) efficient multiplier based on an intermediate hybrid signed-digit (hsd-1) representation," in *Proceedings of the 14th IEEE Sympsoium on Computer Arithmetic, Adelaide, Australia*, pp. 216–224, Apr. 1999.

[38] J. U. Ahmed and A. A. S. Awwal, "Multiplier design using RBSD number system," in *Proceedings of the IEEE 1993 National Aerospace and Electronics Conference*, pp. 180–184, May 1993.

[39] W. Balakrishnan and N. Burgess, "Very-high-speed VLSI 2s-complement multiplier using signed binary digits," *IEE Proceedings-E, Computers and Digital Techniques*, vol. 139, pp. 29–34, Jan. 1992.

[40] D. L. Harris, S. F. Oberman, and M. A. Horowitz, "SRT division architectures and implementations," in *Proceedings of the 13th IEEE Sympsoium on Computer Arithmetic, Asilomar, California, USA*, pp. 18–25, July 1997.

[41] A. M. Nielsen, D. W. Matula, C. N. Lyu, and G. Even, "An IEEE compliant floating-point adder that conforms with the pipelined packet-forwarding paradigm," *IEEE Transactions on Computers*, vol. 49, pp. 33–47, Jan. 2000.

[42] H. Edamatsu, T. Taniguchi, T. Nishiyama, and S. Kuninobu, "A 33 MFLOPS floating point processor using redundant binary representation," in *IEEE International Solid-State Circuits Conference, Digest of Technical Papers*, Feb. 1988.

[43] V. Piuri and R. Stefanelli, "Use of redundant binary representation for fault-tolerant arithmetic array processors," in *Proceedings of the 1988 IEEE International Conference on Computer Design: VLSI in Computers and Processors, ICCD'88*, pp. 496–501, Oct. 1988.

[44] P. A. Ramamoorthy, B. Potu, and G. Govind, "DSP system architecture using signed-digit number represnetation," in *International Conference on Acoustics, Speech, and Signal Processing, ICASSP-88.*, vol. 3, pp. 1702–1705, Apr. 1988.

[45] S.-G. Chen, "A unified bit-parallel arithmetic processor using redundant binary representation," in *Proceedings of the 8th Annual International Phoenix Conference on Computers and Communications*, pp. 91–96, Mar. 1989.

[46] A. Saed, M. Ahmadi, and G. A. Jullien, "Arithmetic with signed analog digits," in *Proceedings of the 14th IEEE Sympsoium on Computer Arithmetic, Adelaide, Australia*, pp. 134–141, Apr. 1999.

[47] S. Vassiliadis, E. M. Schwarz, and D. J. Hanrahan, "A general proof for overlapped multiple-bit scanning multiplications," *IEEE Transactions on Computers*, vol. 38, pp. 172–183, Feb. 1989.

[48] C. N. Lyu and D. W. Matula, "Redundant binary booth recoding," in *Proceedings of the 12th Symposium on Computer Arithmetic, Bath, UK*, pp. 50–57, July 1995.

[49] H. Sam and A. Gupta, "A generalized multibit recoding of two's complement binary numbers and its proof with application in multiplier implementations," *IEEE Transactions on Computers*, vol. 39, pp. 1006–1015, Aug. 1990.

[50] S.-M. Yen, C.-S. Laih, C.-H. Chen, and J.-Y. Lee, "An efficient redundant-binary number to binary number converter," *IEEE Journal of Solid-State Circuits*, vol. 27, pp. 109–112, Jan. 1992.

[51] C.-L. Wey, H. Wang, and C.-P. Wang, "A self-timed redundant-binary number to binary number converter for digital arithmetic processors," in *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 386–391, Oct. 1995.

[52] M. Daumas and D. Matula, "Recoders for partial compression and rounding," Research Report No. RR97-01, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, Jan. 1997. Available at http://www.ens-lyon.fr/LIP/Pub/rr1997.html.

[53] M. Daumas and D. Matula, "Further reducing the redundancy of a notation over a minimally redundant digit set," Research Report No. RR2000-09, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, Mar. 2000. Available at http://www.ens-lyon.fr/LIP/Pub/rr2000.html.

[54] H. L. Garner, "A survey of some recent contributions to computer arithmetic," *IEEE Transactions on Computers*, vol. C-25, pp. 1277–1282, Dec. 1976.

[55] W. J. Cody, JR., "Static and dynamic numerical characteristics of floating-point arithmetic," *IEEE Transactions on Computers*, vol. C-22, pp. 598–601, June 1973.

[56] R. P. Brent, "On the precision attainable with various floating-point number systems," *IEEE Transactions on Computers*, vol. C-22, pp. 601–607, June 1973.

[57] J. D. Marasa and D. W. Matula, "A simulative study of correlated error propagation in various finite-precision arithmetics," *IEEE Transactions on Computers*, vol. C-22, pp. 587–597, June 1973.

[58] W. M. McKeeman, "Representation error for real numbers in binary computer arithmetic," *IEEE Transactions on Electronic Computers*, pp. 682–683, Oct. 1967.

[59] G. W. McFarland, *CMOS Technology Scaling and Its Impact on Cache Delay.* PhD thesis, Stanford University, June 1997.

[60] V. G. Oklobdzija, D. Villeger, and S. S. Liu, "A method for speed optimized partial product reduction and generation of fast parallel multipliers using an algorithmic approach," *IEEE Transactions on Computers*, vol. 45, pp. 294–306, Mar. 1996.

[61] P. M. Farmwald, *On the Design of High Performance Digital Arithmetic Units.* PhD thesis, Stanford University, Aug. 1981.

[62] N. T. Quach, *Reducing the latency of floating-point arithmetic operations.* PhD thesis, Stanford University, Dec. 1993.

[63] J. D. Bruguera and T. Lang, "Leading-one prediction with concurrent position correction," *IEEE Transactions on Computers*, vol. 48, pp. 1083–1097, Oct. 1999.

[64] N. Quach and M. J. Flynn, "Leading one prediction—implementation, generalization, and application," Technical Report No. CSL-TR-91-463, Computer Systems Laboratory, Stanford University, Mar. 1991.

[65] N. Quach and M. J. Flynn, "An improved algorithm for high-speed floating-point addition," Technical Report No. CSL-TR-90-442, Computer Systems Laboratory, Stanford University, Aug. 1990.

[66] S. F. Oberman and M. J. Flynn, "Reducing the mean latency of floating-point addition," *Theoretical Computer Science*, vol. 196, pp. 201–214, 1998.

[67] A. Beaumont-Smith, N. Burgess, S. Lefrere, and C. C. Lim, "Reduced latency IEEE floating-point standard adder architectures," in *Proceedings of the 14th IEEE Sympsoium on Computer Arithmetic, Adelaide, Australia*, pp. 35–42, Apr. 1999.

[68] P.-M. Seidel and G. Even, "On the design of fast IEEE floating-point adders," in *Proceedings of the 15th IEEE Sympsoium on Computer Arithmetic, Vail, Colorado, USA*, pp. 184–194, July 2001.

[69] M. S. Schmookler and K. J. Nowka, "Leading zero anticipation and detection – a comparison of methods," in *Proceedings of the 15th IEEE Sympsoium on Computer Arithmetic, Vail, Colorado, USA*, pp. 07–12, July 2001.

[70] H. A. H. Fahmy, A. A. Liddicoat, and M. J. Flynn, "Parametric time delay modeling for floating point units," in *The International Symposium on Optical Science and Technology, SPIE's 47th annual meeting (Arithmetic session), Seattle, Washington, USA*, July 2002.

[71] P.-M. Seidel and G. Even, "How many logic levels does floating-point addition require?," in *Proceedings of the International Conference on Circuit Design*, pp. 142–149, 1998.

[72] A. M. Nielsen, D. W. Matula, C. N. Lyu, and G. Even, "Pipelined packet-forwarding floating point: II. an adder," in *Proceedings of the 13th IEEE Sympsoium on Computer Arithmetic, Asilomar, California, USA*, pp. 148–155, July 1997.

[73] G. W. Bewick, *Fast Multiplication Algorithms and Implementation.* PhD thesis, Stanford University, Feb. 1994.

[74] P. Bonatto and V. G. Oklobdzija, "Evaluation of booth's algorithm for implementation in parallel multipliers," in *Twenty-nineth Asilomar Conference on Signals, Systems, and Computers, Asilomar, California, USA*, vol. 1, pp. 608–610, Oct. 1995.

[75] W. J. Paul and P.-M. Seidel, "On the complexity of booth recoding," in *Third Real Numbers and Computers Conference Paris, France*, pp. 199–217, Apr. 1998.

[76] H. A. Al-Twaijry, *Area and Performance Optimized Multipliers.* PhD thesis, Stanford University, Aug. 1997.

[77] M. R. Santoro, *Design and Clocking of VLSI Multipliers.* PhD thesis, Stanford University, Oct. 1989.

[78] H. A. Al-Twaijry and M. J. Flynn, "Optimum placement and routing of multiplier partial product trees," Technical Report No. CSL-TR-96-706, Computer Systems Laboratory, Stanford University, Sept. 1996.

[79] J. Um and T. Kim, "An optimal allocation of carry-save-adders in arithmetic circuits," *IEEE Transactions on Computers*, vol. 50, pp. 215–233, Mar. 2001.

[80] H. A. Al-Twaijry and M. J. Flynn, "Technology scaling effects on multipliers," *IEEE Transactions on Computers*, vol. 47, pp. 1201–1215, Nov. 1998.

[81] A. A. Liddicoat and M. J. Flynn, "Parallel computation of the square and cube function," Technical Report No. CSL-TR-00-808, Computer Systems Laboratory, Stanford University, Aug. 2000.

[82] E. Antelo, M. Bóo, J. D. Bruguera, and E. L. Zapata, "A novel design of a two operand normalization circuit," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 6, pp. 173–176, Mar. 1998.

[83] J. Cortadella and J. M. Llabería, "Evaluation of $a+b = k$ conditions without carry propagation," *IEEE Transactions on Computers*, vol. 41, pp. 1484–1488, Nov. 1992.

[84] B. Parhami, "Comments on "evaluation of $a + b = k$ conditions without carry propagation"," *IEEE Transactions on Computers*, vol. 43, p. 381, Apr. 1994.