

INCORPORATING MULTIPLICATION INTO DIGIT-
RECURRENCE DIVISION AND THE SQUARE ROOT

TO CALCULATE $\frac{A \times B}{D}$, $\sqrt{A \times B}$

BY

ORWA MU'MEN DIRANEYYA

A Thesis Presented to the
DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

COMPUTER ENGINEERING

December 18, 2012

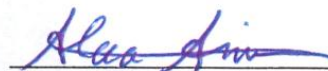
KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS

Dhahran, Saudi Arabia

DEANSHIP OF GRADUATE STUDIES

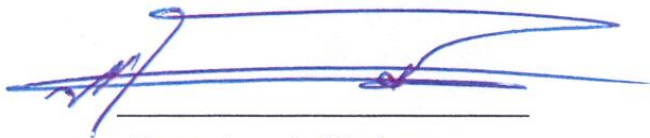
This thesis, written by *Orwa Mu'men Diraneyya* under the direction of his thesis advisor and approved by his thesis committee, has been presented to and accepted by the *Dean of Graduate Studies*, in partial fulfillment of the requirements for the degree of MASTER OF SCIENCE in COMPUTER ENGINEERING.

Thesis Committee

 Dec. 22, 2012

Thesis Advisor

Dr. Alaaeldin Amin



Department Chairman

Dr. Basem Al-Madani



Member

Dr. Abdelhafid Bouhraoua

 30/12/12

Dean of Graduate Studies

Dr. Salam A. Zummo



 Dec. 22, 2012

Member

Dr. Aiman El-Maleh

Incorporating Multiplication into
Digit-Recurrence Division and the Square Root
to Calculate $\frac{A \times B}{D}$, $\sqrt{A \times B}$

By

Orwa Mu'men Diraneyya

A Thesis Presented to the

DEANSHIP OF GRADUATE STUDIES

in Partial Fulfillment of the Requirements, for the degree of

MASTER OF SCIENCE

IN

COMPUTER ENGINEERING

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS

Dhahran, Saudi Arabia

December 29, 2012

*This thesis is dedicated to my mom, Mu'mena Fadeel Al-Mu'ayyad,
for life, culture and relatives have never done her justice*

Acknowledgement

Finishing this thesis while working in Dubai as a barista, was a great challenge that I could not have faced on my own. That is to say, there was an interesting personal story behind this thesis, with many people taking a good part in it... People without whom I would not have eventually succeeded.

The full names of some of those people I did not know, but I'm so grateful to them that I will try to mention them all in no particular order, sometimes using their first names only, along with the type of help they offered me.

To start with, I thank my advisor, Dr. Alaaeldin Amin, who's rare and unstoppable support is the main reason why I am here again in Saudi Arabia, after one year of leaving this country, to defend my thesis.

He did not judge me for going to Dubai to work in a restaurant, he understood that I had personal problems, he was almost always understanding and considerate to my needs. His flexibility, positivity, open-mindedness, and willingness to bend for me is immensely appreciated. He never told me I was doing poorly (even when I was), knowing how fragile I was at times. Rather, he always emphasized the bright side of things.

For him I will do anything. I wanted to succeed to make myself proud, but the only person beside myself that I really wanted to make proud was him.

I'd like also to thank the rest of the committee members: Dr. Abdelhafid Bouhraoua and Dr. Aiman El-Maleh.

After that, in no particular order, I'd like to thank Wendy Mashkind who helped me on a personal level. She is my pen pal and close friend, which I never met.

Note that a wise man, beside knowing who to thank, knows who not to, and in my

case this would be the RAW coffee company in Dubai, who helped me find my first job in Galeries Lafayette. I also choose not to thank Galeries Lafayette or any of the coworkers there. Yet, I choose to thank Simon, the CEO, for being a reasonable and a respectable man. He had to fire me under the pressure of a powerful local guest, but I did not hate him for it.

Speaking of my coworkers in Galeries Lafayette, there were few exceptions. Namely, I thank Kerryon Romeril for many things. I thank her for being extremely friendly and genuine, unlike most of the others, and for selling me and giving me many of the things that I still possess today, including my beautiful green carpet.

I'd like also to thank all of my coworkers in Park Central, the restaurant where I am working right now, for being the sweetest coworkers someone could ever have. I thank them for all the encouragement, all the care, and all the nice words. I also thank my boss, Talal Thabet, for giving me a full week off (with no questions asked), at the time when that one week meant either succeeding or failing this Masters.

I also thank him for providing me with a great working environment, which as I realize today, is the best working environment I could get in Dubai while pursuing my passion in coffee. Thank you for that, Talal, and I wish that Park Central will eventually succeed like I did!

I also thank Helen, a temporary British manager at Park Central who reduced the working hours from 10 to 9 hours, as well as giving me two days off per week during the month of Ramadan. That really helped. The same also goes for our current manager, Mr. Nadeem, who has given me a couple of days off in the last weeks. Thank you Nadeem.

As for obtaining a Visa to Saudi Arabia, my utmost gratitude goes to Muntaser Diraneyya, Munjed Diraneyya, and my father for helping me with the Visa. I also thank Amr Hatahet. Embarrassingly, an army of men had to work day and night to make the Visa that allowed me to come to this country for my defense, I thank you all! Without your help this defense would not have not been possible!

I also thank Mr. Jarrah Assaleem from the Saudi Embassy in Dubai, and all of the others working there, for being extremely understanding, very respectful, and so kindly

helpful in getting me a Visa on the same day, and allowing me to apply for it myself from the embassy, which is usually not permitted. I am so grateful to you guys!

I also thank Anas Ghazal, my cousin, for accommodating me while in Saudi Arabia. He provided me with a room 10 times the size of my room, a desk that I did not have in Dubai, and good company. I am so grateful to you Anas!

There has also been other people who helped me in the last year, and one of those is Ahmad Zahidah, who helped me financially. Many things would have went totally wrong without this help, Ahmad, and I am going to pay you back immediately after I get back, I promise!

I also want to thank Rachell Enson, for being one true friend of mine in this bad time of my life. Rachell, you are always on my mind.

Last but not least, I thank Adam Chalk for his friendship, his encouragement, and his continual support. Thanks Adam.

Table of Contents

List of Tables	ix
List of Figures	x
Abstract (English)	xii
Abstract (Arabic)	xiii
1 Introduction	1
1.1 Literature Review	1
2 Theory of Digit-Recurrence Multiplication/Square Root	6
2.1 Iterative Square Root	6
2.1.1 Convergence inequality (theoretical)	7
2.1.2 Implied radicand function	8
2.1.3 Convergence inequality (practical)	10
2.1.4 The resulting scheme	10
2.1.5 Meaning of the remainder R_j	11
2.2 Subtractive Square Root	11
2.2.1 Adjusting the square root \mathcal{S}_{i-1}	11
2.2.2 Maintaining the remainder R_{i-1}	11
2.2.3 Initializing the remainder R_0	13
2.2.4 The resulting method	13
2.3 Long Square Root	14
2.3.1 Order of digit extraction	15
2.3.2 Digits as adjustments	16
2.3.3 Resulting algorithm	16
2.3.4 Correctness vs. increased accuracy	17
2.3.5 Paper-and-pencil method	19
2.3.6 Putting it all together	23
2.4 Incorporating Multiplication	24

2.4.1	An approximate radicand \tilde{N}_i	25
2.4.2	Precision requirements of \tilde{N}_i	26
2.4.3	The direct approach	28
2.4.4	An alternative approach.....	34
2.4.5	The Z parameter	37
2.5	Finding a Recurrence Relation	39
2.5.1	A recurrence relation for updating \mathcal{S}'	40
2.5.2	A recurrence relation for updating \tilde{R}'	41
2.5.3	Putting it all together	44
3	SRT Table Design	46
3.1	A Recurrent Selection Criterion	47
3.2	The P - $\dot{\mathcal{S}}$ Plane	51
3.2.1	Feasibility	51
3.2.2	Comparison constants	53
3.2.3	Iteration dependence.....	58
3.2.4	Determination of the constants	60
3.2.5	Evolving precision of $\dot{\mathcal{S}}$ (Advanced)	65
3.2.6	Unsigned SRT table (Advanced)	70
3.2.7	Custom mappings (Advanced).....	71
3.3	Sharing the SRT Table with Division	72
3.3.1	Effect on $n_{Pintegral}$	74
3.4	Generating the SRT table contents	74
3.4.1	A design example	75
3.5	The Software Platform	78
3.5.1	Exploration Strategy.....	79
3.5.2	Good Designs	80
4	Hardware Design	82
4.1	A Register-Based Algorithm Description	82
4.1.1	Register charts	84
4.2	Loose Leading Bit of $\dot{\mathcal{S}}$	86
4.2.1	Range of $\dot{\mathcal{S}}$ in a square-root unit	86
4.2.2	Range of $\dot{\mathcal{S}}$ in the fused unit	87
4.2.3	Alternative addressing scheme.....	89
4.2.4	Design of the First-Digit Selector	91
4.2.5	Analytical minimum bound on Z	97
4.2.6	Aided exclusion of digit choices	98
4.3	The Selection Inputs: P and $\dot{\mathcal{S}}$	99

4.3.1	Relationship between P_{i-1} and W_{i-1}	99
4.3.2	Sampling P from the W register	100
4.3.3	Relationship between \hat{S}_{i-1} and S_{i-1}	101
4.4	Number of iterations	103
4.4.1	Final accuracy	103
4.4.2	Truncated accuracy	104
4.4.3	Rounded accuracy	104
4.4.4	Computing the number of iterations	105
4.5	Hardware Functions	105
4.5.1	Updating the result S register	105
4.5.2	Formation of the $S0s$ part	107
4.5.3	Updating the residual W register	108
4.5.4	The digit-selection path	111
4.6	Optimizing the Critical Path	113
5	Results and Discussion	115
5.1	System Fingerprint	116
5.1.1	Simulation Output	116
5.2	Comparisons and Conclusion	121
5.3	Future Work	122
A	Theory of Digit-Recurrence Multiplication/Division*	123
A.1	The Recurrence Relation	123
A.1.1	An additive algorithm	124
A.1.2	Incorporating multiplication	125
A.2	The Correctness Constraint	127
	Bibliography	133
	Vita	136

List of Tables

3.1	The SRT table contents of a maximally-redundant, radix-4 system	76
3.2	List of system parameters leading to a good design of the table	81
4.1	Design data of the First-Digit Selector in a radix-4 system	96
4.2	Design data of the First-Digit Selector in a radix-4 system (carry-save)	96
4.3	P sampling bit-position sequence	100
4.4	On-the-fly conversion of digits	107

List of Figures

2.1	$\frac{error(\mathcal{N})}{N}$ vs. $\frac{error(\mathcal{S}/\mathcal{Q})}{S/Q}$	9
2.2	Updating the remainder	12
2.3	Digits as adjustments	15
2.4	Digit-selection timeline.....	17
2.5	Bare result $\bar{\mathcal{S}}_{i-1}$	21
2.6	Positioning of the linear-quadratic term	22
2.7	Two stages of the fused algorithm	29
2.8	Delayed square root \mathcal{S}'	37
2.9	The Z parameter	37
2.10	Shifting the remainder between iterations	41
2.11	Shifting the remainder for two terms with mismatching pace	42
3.1	Fractional result $\dot{\mathcal{S}}_{i-1}$	49
3.2	Feasibility categories	52
3.3	Effective height of the overlap region.....	54
3.4	Overlap region's effective existence	54
3.5	Comparison constants	55
3.6	P representation and truncation intervals	61
3.7	Minimum effective height of the overlap regions	62
3.8	Codomain of the comparison constants	62
3.9	Preliminary, early, and late iterations	66
3.10	Inclusion/exclusion of intervals	69
3.11	Folded overlap region	70
3.12	Hardware needed for an unsigned SRT table.	71
3.13	Mapping logic	72
3.14	Custom-mappings interface.....	72
3.15	System parameter view of maximally-redundant, radix-4 system	77
3.16	Designer cell view of maximally-redundant, radix-4 system	77
3.17	SRT table design flow	79
3.18	Strategy for SRT-table design space exploration.....	80

4.1	Problematic range of $\dot{\mathcal{S}}$	88
4.2	Tight overlap regions associated with the result's extended range	88
4.3	Dynamic normalization of truncated result	92
4.4	Circuit for First-Digit Selector	95
4.5	Support circuit for the First-Digit Selector in Table 4.2	97
4.6	\hat{P} total bit count	101
4.7	Circuit for the iteration-indicator register.	102
4.8	Circuit for sampling P at a changing bit position	102
4.9	The $\dot{\mathcal{S}}$ register design.	103
4.10	Circuit for converting the result digits on-the-fly	107
4.11	Circuit for forming the $\mathbf{S0s}$ part on the fly	108
4.12	Circuit of the \mathbf{B}^{\rightarrow} register	109
4.13	Circuit for adding the partial-product term	110
4.14	Relative positioning of the partial-product term	111
4.15	Relative positioning of the linear-quadratic term	111
4.16	Circuit for subtracting the linear-quadratic term	111
4.17	Circuit for the digit-selection path	112
4.18	Complete circuit for a maximally-redundant, radix-4 system	114

Thesis Abstract

Full Name of the Student

Orwa Mu'men Diraneyya

Title of the Study

*Incorporating Multiplication into Digit-Recurrence Division and
“ the Square Root to Calculate $\frac{A \times B}{D}$, $\sqrt{A \times B}$ ”*

Major Field

Computer Arithmetic

Date of Degree

2012

This thesis discusses the theory and the design of a novel, multiplier-free unit that computes $\sqrt{A \times B}$. The fused operation is achieved through a single digit-recurrence relation, using an initial-delay-like parameter that is fixed at design time. The thesis also elaborates on the theory of a fused unit that computes $\frac{A \times B}{D}$, which is already proposed in the literature, while addressing the desire to share the SRT table between the two units.

ملخص الرسالة

الاسم

عروة مؤمن مأمون ديرانية

عنوان الرسالة

دمج الضرب في عملية القسمة والجذر التربيعي

التخصص

الحسابات الرقمية

تاريخ التخرج

٢٠١٢

تتناول هذه الرسالة دمج عملية الضرب بعملية الجذر التربيعي، وهو ما ينتج عن خوارزمية غير معروفة قبلاً. كما وتحدث الرسالة أيضاً عن دمج عملية الضرب بعملية القسمة، وعن كيفية مشاركة بعض المكونات بين الوحدتين، وهو ما من شأنه تقليل الكلفة العتادية لهاتين الداراتين الحسابيتين.

Chapter 1

Introduction

1.1 Literature Review

In the literature, the SRT square root seems to be mostly seen as an extension of SRT division, which gets most of the emphasis. This trend is encouraged by the instruction-count statistics which show that the performance of systems is not affected by the performance of the square-root unit to the extent it is affected by the performance of division, suggesting that the least-performing square-rooting technology (a radix-2 unit) would usually be sufficient [19]. This leads to most of the reviewed material here discussing SRT division, not the square root, but with the discussions often being equally-applicable to the case of the square root.

For example, a paper discussing on-the-fly conversion of the quotient digits would be equally-applicable to the case of the root digits, since both algorithms produce the result digits in a redundant form. Having said that, even though the review will often be referring to SRT division, it must be understood that in many cases, the discussions would also apply to the case of the square root.

The SRT division was given this name by Freiman [12], by combining the initials of the three people who seemed to discover it independently at about the same time, which are D. Sweeney of IBM, J. E. Robertson of the University of Illinois, and T. D. Tocher.

Since the introduction of the use of redundant representations for the remainders

by D. E. Atkins [3], the SRT method has been studied extensively and implemented in processors. A survey shows that out of 13 recent processors, 11 use SRT division for performing floating-point division [18].

Unlike the restoring and the non-restoring methods [4], the SRT method is characterized by the nature of the quotient digits as produced during division, and by the way in which such digits are chosen [23].

Because of the redundant nature of the chosen digits, conversion of the quotient to the conventional representation is required for this class of division methods, which is usually done serially [9] (thus avoiding an expensive carry-propagate operation at the end of division).

Unlike the restoring and the non-restoring methods, where the selection of the next result digit is guided by the sign of the previous remainder, the SRT method is distinguished by its ability to select the next result digit based on the value of the remainder. Robertson [24] suggested that the mechanism for selecting the next result digit be viewed as a “limited-precision model” of the full-precision division (i.e. a *minute division* step). Atkins [3], on the other hand, contemplates the nature of such mechanism, suggesting that it can fall in one of two broad categories:

- Either be arithmetic (i.e. the “arithmetic models”): which refer to the use of arithmetic operations to obtain the correct quotient digit from the value of the current remainder.
- Or be static (i.e. the “table look-up models”): which refer to the use of the logical equivalent of a look-up table, to obtain the correct quotient digit from the value of the current remainder.

In the overwhelming majority of designs, this mechanism is implemented as a lookup table, which is known as the SRT table.

Note that a relatively-recent PhD thesis [18] from the University of Adelaide suggests that the arithmetic model be reconsidered for implementing the digit-selection function (in Chapter 4: *Comparison Multiples, a Different Approach to Quotient Digit Selection*).

In this approach, called the comparison multiples, limited-precision multiples of the divisor are calculated prior to division, then a carry-free subtraction is used along with sign detectors to compare the current remainder against each one of these multiples, with the signals passed through a priority encoder to find a correct choice of the next quotient digit.

Unlike this atypical approach, this thesis assumes that the digit-selection function is to be implemented using a PLA (a two-level logical implementation, signifying a look-up function). The reason behind this choice is that this is the most common way of implementing the digit-selection function today, and the one that results in the minimum delay for the digit-selection step (allowing for a shorter cycle time).

In fact, much of the research in SRT algorithms has been dedicated to reducing the complexity of this table, using a variety of techniques that reduce the size of the table at the cost of adding some external logic [5][15][8][26].

To design the contents of the table, we need to find a set of truncation parameters n_P and n_D (also called the *truncation pair*) that allows a correct selection of the next quotient digit. Robertson, in his paper back in 1985 [24], carried out a simple theoretical analysis assuming that $n_P = n_D$ (which is not usually the case), which resulted in truncation values higher (i.e. worse) than what is actually necessary. Atkins, in his 1968 paper [3], attempted a theoretical analysis based on extrapolating the required precision from the height of the overlap region at the minimum value of the divisor (assuming that the required precision is a linear function of the divisor value). His results, however, predict less precision than what is actually needed for the partial remainder and the divisor [25]. In [26], Harris, Oberman and Horowitz analyze, based on the use of Taylor diagrams, the relationship between the required precision in the partial remainder n_P and in the divisor n_D . They also consider the case of a remainder in the carry-save form in which the truncation precision of the carry component n_{PC} need not be the same as that of the sum component n_{PS} .

To ease looking for a valid truncation pair (values for n_P and n_D), Parhami reduces the search space to four truncation pairs in [21], while using an exhaustive test (a nested

program loop that goes through all the overlap regions) to check the feasibility of each one of the four truncation pairs. In the conclusion section of his paper, Parhami comments that it is of interest to come up with simple analytical tests (as opposed to exhaustive search) to check the feasibility of each one of the four truncation pairs, however, he comments that it's only of theoretical interest to do so as the algorithm provided for exhaustive testing is simple enough for practical purposes.

Kornerup, in his paper in 2003 [15], proposes a refined method for finding the truncation parameters, which further optimizes the exhaustive testing step of Parhami's scheme. Further, he generalizes the method to the case of the SRT square root in his other, 2005 paper [16].

For the reason mentioned in Parhami's paper, coupled with the abundance of computational power today, this thesis shows little interest (if any) in the analytical approaches to finding good values of n_P and n_D/n_S , which is the trend signified by the SRT-table software tools developed as part of this thesis.

In the same paper, Kornerup points out the symmetry in the P-D diagram of SRT division, and proposes a scheme for exploiting this symmetry, thus cutting down the size of the SRT table in half, at the cost of adding some logic external to the table. Note that an earlier attempt to exploit the same size-reducing possibility is found in the US Patent entitled "Simplification of Lookup Table" [5]. In this patent, a carry-save representation of the remainder is assumed, and a negative-to-positive mapping based on the 2's-complement is suggested. However, since the 2's complement is a carry-propagate operation, this solution is inferior to Kornerup's proposal in his 2003 paper [15].

Kornerup, on the other hand, suggests a smart workaround by negating the truncated carry and sum components prior to adding them up to form the estimate, which results in the $-2ulp$ translation shift needed to ensure a proper negative-to-positive mapping of the "uncertainty rectangle" in the case of a carry-save remainder. Note that in this case, the sign of the resulting digit will be determined by the sign of the remainder, since the SRT table will not store any negative digit values.

In the case of the SRT square root, the symmetry does not hold in early iterations of

the algorithm. This iteration dependence in the case of the square root has traditionally pushed the designers to use an initial PLA, which adds to the hardware complexity of the algorithm. To avoid this, Ercegovac [8] shows that by placing well-designed logical transformations at the entry of the SRT table, we might be able to lessen the affect of this dependence on the selection ranges, which results in an iteration-independent look-up table in the radix-4 case. Note that some other ideas and theoretical refinements also play a role in enabling the design of this iteration-independent look-up table, which has greatly inspired the development of the *selection function designer* tool in the section 3.5.

In [14], an algorithm for division with both an on-line dividend and an on-line divisor is developed, with ideas very similar to the ones used in the theoretical development of this thesis, leading to the multiplication-enhanced algorithms.

In [20], a methodology and tools similar to the ones presented here are developed and used to measure the complexity of the SRT table in SRT division, by counting the number of product terms in the minimized logic of the table.

Unlike those *undisclosed* tools, the tools of this thesis are well-documented and fully disclosed, with features permitting the design of higher-radix systems.

To combine the digit selection of SRT division and the square root, some original research has been conducted by the student as part of the COE606 *Independent Research* course in 2009, which is also utilized in this research [7, 6].

Chapter 2

Theory of Digit-Recurrence Multiplication/Square Root

2.1 Iterative Square Root

Finding the square root is the *inverse* of computing the square:

$$A \begin{array}{c} \xrightarrow{\text{compute square}} \\ \xleftarrow{\text{find square root}} \end{array} A^2$$

Hence, to compute the square root we assume that the number in hand (equal to A^2 above, called the *radicand*), is already the result of squaring some other quantity, which we refer to as the *square root* A .

Throughout this thesis, we will be using the letters N and S , respectively, to denote the radicand and the square root of an arbitrary square-root problem:

$$\boxed{S = \sqrt{N}}$$

where: $N = S^2$

Unfortunately, there aren't many ways to describe S beside being the amount whose square is equal to the radicand N . The fact S is specified indirectly (as the number

meeting the criterion above) implies that it has to be approached through search (or developed in steps), suggesting an *iterative* algorithm.

Furthermore, due to the nature of modern numbers, radicands with limited or few digits are often blessed with a square root that has an unlimited number of digits:

$$\sqrt{\underbrace{\square\square\square\square}_{\text{few digits}}} = \underbrace{\square\square\square\square\square}_{\text{endless digits}}\cdot$$

This has an important impact on a computational unit for calculating the square root, as the exact (i.e. endless) square root is not a feasible outcome of such unit. Rather, there exists an inherent compromise where:

calculation time/effort	\propto	precision of the result
----------------------------	-----------	----------------------------

This, along with the previous suggestion that a machine implementation of the square root would be iterative, explains why all the digital methods for finding the square root rely on the principle of repeatedly correcting/enhancing an approximation \mathcal{S}_i , until an exact or a sufficiently-accurate result \mathcal{S}_n is obtained:

$$\begin{array}{ccc} \text{initial} & & \text{the} \\ \text{approximation} & & \text{result} \\ \downarrow & & \downarrow \\ \mathcal{S}_0: \mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3, \mathcal{S}_4, \dots \mathcal{S}_{n-1}, \mathcal{S}_n \\ \underbrace{\hspace{10em}}_{n \text{ iterations}} \end{array}$$

In such a digital unit, the calculation effort is interpreted as the number of iterations, which is also proportional to the latency of the unit.

2.1.1 Convergence inequality (theoretical)

For a meaningful iteration, the approximate square roots \mathcal{S}_i must be increasing in accuracy, or in a mathematical language, must be converging to the true square root S :

$$|S - \mathcal{S}_0| > |S - \mathcal{S}_1| > |S - \mathcal{S}_2| > |S - \mathcal{S}_3| > \dots > |S - \mathcal{S}_n|$$

Unfortunately, this test cannot be implemented in practice, since the value of S is unknown. Hence, a guiding metric other than the *direct error* is required.

2.1.2 Implied radicand function

To create practical means of guiding the convergence of the result, we modify the definition criterion in a simple, but significant way:

$$N = S^2 \quad \xrightarrow{\text{becomes}} \quad \begin{array}{c} \text{(2) radicand} \\ \text{implied by } \mathcal{S} \\ \downarrow \\ \mathcal{N} = \mathcal{S}^2_{\square} \\ \text{(1) an arbitrary} \\ \text{approximation} \end{array}$$

Our goal is to use this *implied-radical* function $\mathcal{N}(x) = x^2$ to guide the convergence of the result. To do this, we pass an approximation of the square root \mathcal{S} as well as the true square root S to this function:

$$\mathcal{N}(\mathcal{S}) = \mathcal{S} \times \mathcal{S}$$

$$\mathcal{N}(S) = S^2 = N$$

Since both values are either directly given (N) or can be digitally computed ($\mathcal{S} \times \mathcal{S}$), the difference can be obtained digitally (called the error in the implied radicand):

$$error(\mathcal{N}) = \mathcal{N}(S) - \mathcal{N}(\mathcal{S}) = S^2 - \mathcal{S}^2$$

Now remembering that

$$error(\mathcal{S}) = S - \mathcal{S}$$

it is natural to ask whether the error in the implied radicand can be used as an *indirect measure* of the error in \mathcal{S} , which requires the following:

- That both errors have the same polarity (to drive the sign of the next adjustment).
- That both errors be correlated (to drive the magnitude of the next adjustment).

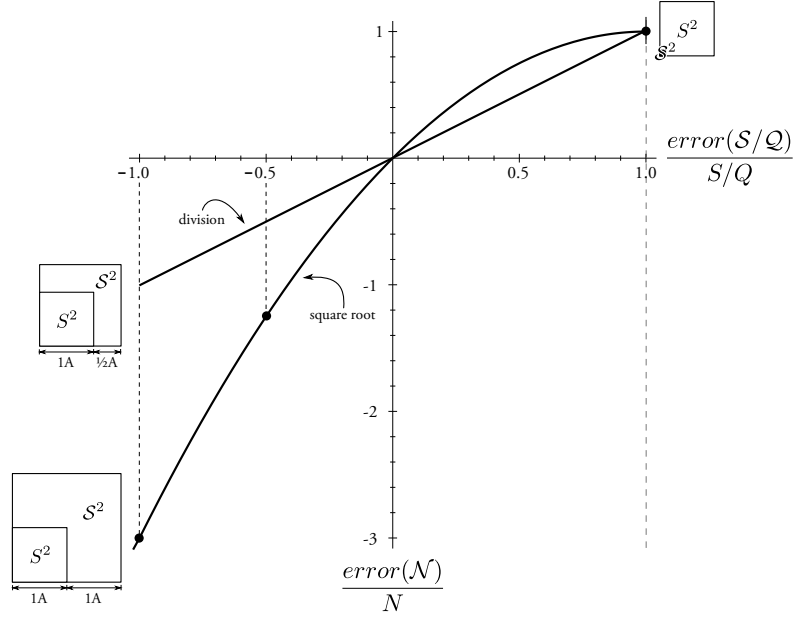


Figure 2.1: Error in the implied radicand/dividend versus error in the square root/quotient

To see if the new error value answers those demands, we express the error in \mathcal{N} as a function of the error in \mathcal{S} , to plot one against the other. Unfortunately, the resulting expression is not only a function of the error in \mathcal{S} , but also a function of S (i.e. problem-dependent), and hence cannot be translated to a simple 2-dimensional plot (i.e. a plot of $error(\mathcal{N})$ vs. $error(\mathcal{S})$).

To overcome this technical difficulty, we express the *relative indirect error* $\frac{error(\mathcal{N})}{N}$ as a function of the *relative direct error* $\frac{error(\mathcal{S})}{S}$, which results in a single-variable expression that can be easily plotted (Fig. 2.1).

Looking at the figure, not only that the polarities of the two errors are matched, it also shows that the error in the implied radicand responds with higher sensitivity to the deviation of \mathcal{S} from S , if compared to the error in the *implied dividend* in the case of division (which, on the other hand, responds to the deviation of the approximate quotient \mathcal{Q} from the true quotient Q —more on this in Appendix A).

Hence, the error in the implied radicand seems to be just as good of an indicator of the actual error as the error in the implied dividend (in the case of division), implying that the error in the implied radicand \mathcal{N} can be successfully used as the basis of an iterative scheme for finding the square root.

2.1.3 Convergence inequality (practical)

In a practical implementation of the square root, error in the implied radicand

$$error(\mathcal{N}) = N - \mathcal{S}^2$$

can be used both to guide and to supervise the development of the result, through enforcing the following inequality:

$$|N - \mathcal{S}_0^2| > |N - \mathcal{S}_1^2| > |N - \mathcal{S}_2^2| > |N - \mathcal{S}_3^2| > \dots > |N - \mathcal{S}_n^2|$$

The same inequality can be concisely written as:

$$|R_0| > |R_1| > |R_2| > |R_3| > \dots > |R_n|$$

where each one of the error values is called the *remainder*, and is maintained throughout the algorithm⁽¹⁾.

2.1.4 The resulting scheme

$$\begin{aligned} \mathcal{S}_0 &= \textit{initial approx.} \\ 1 \leq i \leq n : \quad \mathcal{S}_i &= \textit{enhance}(\mathcal{S}_{i-1}, R_{i-1}) \quad \text{such that } |R_i| < |R_{i-1}| \quad (2.1) \\ (R_j &\stackrel{\text{def}}{=} N - \mathcal{S}_j^2) \end{aligned}$$

In this scheme for finding the square root, an initial approximation of the square root is repeatedly adjusted, guided by the amplitude and the sign of the current remainder R_{i-1} , through the repeated application of the function *enhance*.

The objective of the *enhance* function is thus to produce better approximation of the square root in every iteration, which is guaranteed by the practical condition $|R_i| < |R_{i-1}|$.

⁽¹⁾ This is the parallel amount of the *division remainder*, which is different from what will be referred to later as the *residual*.

2.1.5 Meaning of the remainder R_j

The remainder R_j contains what remains of the radicand $N = S^2$ after extracting the square of the corresponding root \mathcal{S}_j^2 from it, hence the name “remainder”. The name also signifies the desire to reduce this amount without limit, which corresponds to \mathcal{S}_j indefinitely approaching S (the true square root).

2.2 Subtractive Square Root

In this section, we’ll define a variant of the iterative square root where additions and subtractions are the only mathematical operations allowed as part of the execution of the algorithm. This means that only these two operations are allowed to:

1. Maintain the remainder R_{i-1} (needed for determining the next adjustment).
2. Adjust the square root \mathcal{S}_{i-1} .

Note that determining the next adjustment is not discussed as part of the subtractive square root. This makes the subtractive square root an *incomplete algorithm*, or a class of algorithms from which a practical algorithm can be derived.

2.2.1 Adjusting the square root \mathcal{S}_{i-1}

To meet the criterion of the subtractive square root, we have no other option but to enhance the square root through discrete, additive adjustments $\Delta\mathcal{S}_i$:

$$\mathcal{S}_i = \mathcal{S}_{i-1} + \Delta\mathcal{S}_i$$

2.2.2 Maintaining the remainder R_{i-1}

To maintain the remainder in an iterative machine implementation, it has to be updated in every iteration to ensure that the residual R_{i-1} of next iteration (i.e. R_i) will have the correct relationship with the enhanced square root \mathcal{S}_i . As mentioned earlier, this has to be done additively, as follows:

$$R_i = R_{i-1} + \Delta R_i$$

where:

$$\begin{aligned}
\Delta R_i &\stackrel{\text{def}}{=} R_i - R_{i-1} \\
&= (N - \mathcal{S}_i^2) - (N - \mathcal{S}_{i-1}^2) \\
&= -(\mathcal{S}_i^2 - \mathcal{S}_{i-1}^2) \\
&= -((\mathcal{S}_{i-1} + \Delta \mathcal{S}_i)^2 - \mathcal{S}_{i-1}^2) \\
&= -((\mathcal{S}_{i-1}^2 + 2\mathcal{S}_{i-1}\Delta \mathcal{S}_i + \Delta \mathcal{S}_i^2) - \mathcal{S}_{i-1}^2) \\
&= -2\mathcal{S}_{i-1}\Delta \mathcal{S}_i - \Delta \mathcal{S}_i^2
\end{aligned}$$

The resulting expression for ΔR_i represents the difference in the square between \mathcal{S}_{i-1} and \mathcal{S}_i due to an additive adjustment of $\Delta \mathcal{S}_i$. (Fig 2.2)

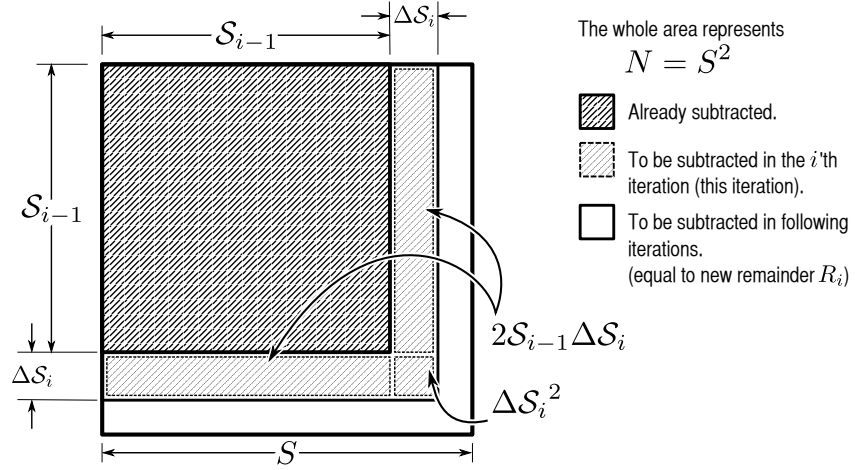


Figure 2.2: A depiction of the terms involved in updating the remainder

To ease referring to each one of these two terms, we choose to name $2\mathcal{S}_{i-1}\Delta \mathcal{S}_i$ the *linear term*, and to name $\Delta \mathcal{S}_i^2$ the *quadratic term*.

2.2.3 Initializing the remainder R_0

In the subtractive square root method, the square root is enhanced from iteration to iteration, while the remainder R_i is also being incrementally maintained, as follows:

$$\begin{aligned}\mathcal{S}_i &= \mathcal{S}_{i-1} + \Delta\mathcal{S}_i \\ R_i &= R_{i-1} - 2\mathcal{S}_{i-1}\Delta\mathcal{S}_i - \Delta\mathcal{S}_i^2\end{aligned}$$

Clearly, this scheme only works if the remainder is initialized correctly, meaning that both \mathcal{S}_0 and R_0 be correctly related as follows:

$$R_0 = N - \mathcal{S}_0^2$$

A natural choice of the initial square root \mathcal{S}_0 , which also results in a computation-less initialization of the remainder, is zero:

$$\mathcal{S}_0 \rightarrow 0 \quad \Rightarrow \quad R_0 \rightarrow N$$

2.2.4 The resulting method

$$\begin{aligned}\mathcal{S}_0 &= 0 \\ R_0 &= N \\ 1 \leq i \leq n : \quad \mathcal{S}_i &= \mathcal{S}_{i-1} + \Delta\mathcal{S}_i \\ R_i &= R_{i-1} - 2\mathcal{S}_{i-1}\Delta\mathcal{S}_i - \Delta\mathcal{S}_i^2\end{aligned} \tag{2.2}$$

Note that without a useful scheme for finding the adjustments $\Delta\mathcal{S}_i$, this description cannot be used as a practical algorithm for the square root.

In this light, the subtractive square root serves the role of an intermediate algorithm in the gradual development of the digit-recurrence algorithm.

2.3 Long Square Root

By looking at (2.2), we can see that the final result \mathcal{S}_n is developed, through the course of n iterations, by accumulating n discrete adjustments:

$$\mathcal{S}_n = \sum_{j=1}^n \Delta \mathcal{S}_j$$

This format for the final output is too general and hence impractical for a digital implementation, as it implies full-length summation. A natural upgrade of the method, hence, would be to substitute more intuitive components of the result for each one of these adjustments.

In a digital computer, binary words similar to the decimal numbers we use everyday are used to convey numbers. This means that in a machine implementation of the square root, the result \mathcal{S}_n is expected to be eventually translated to a sequence of bits, or more generally digits.

Being the most intuitive constituents of a number in computers, it makes a perfect sense to try to confine the adjustments $\Delta \mathcal{S}_i$ to one- or multiple-bit contributions to \mathcal{S}_{i-1} .

Not only that this will make constructing the final result \mathcal{S}_n as straightforward as concatenating the bits/digits contributed by the different adjustments $\Delta \mathcal{S}_i$, but will also set clear bounds on what will constitute a correct and an incorrect choice of bits/digits, as well as setting a standard on how fast/slow the precision of \mathcal{S}_i should be paced from iteration to iteration, and what final precision to expect upon the end of the algorithm.

To achieve all this, we will modify the subtractive square root to retire a fixed number of result bits per iteration. Note that once a number of result bits are retired, they are final and would not be revisited or altered in later iterations.⁽²⁾

The process of electing a correct sequence of m result bits per iteration is referred to as *digit selection*, whereas the process of updating the remainder accordingly is referred to as the *extraction of digits*.

⁽²⁾ This is in contrast with the restoring and the non-restoring division/square-root algorithms, which are variable-time algorithms.

2.3.1 Order of digit extraction

An important aspect of the intended upgrade is the order of determining the result digits, which will influence the order of digit extraction.

Generally speaking, the right order for determining the result digits depend on:

1. The nature of the numbering system used⁽³⁾, and
2. The nature of the mathematical operation being performed.

For example, in the case of additions/subtractions, result digits are determined from the least- to the most-significant. This is unlike the case of division (and the square root, for that matter), where the digits of the result can only be determined from the most- to the least-significant.

A detailed explanation of why this is so is beyond the scope of this thesis, but can generally be achieved by modeling division and the square root as *counting problems*, where multiples of radix powers are to be extracted (one by one) from the whole set, starting from the most-significant, in order to yield a correct count.

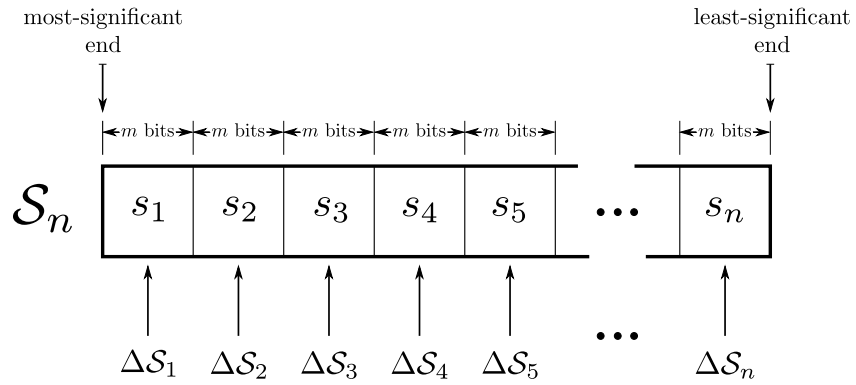


Figure 2.3: Substituting digits of the result for the various adjustments, while adhering to a strict, most-to-least-significant ordering.

⁽³⁾ A radix-based representation in this case.

2.3.2 Digits as adjustments

To confine the adjustments $\Delta\mathcal{S}_i$ to ordered, m -bit revelations of S , we need to apply the following substitution to (2.2):

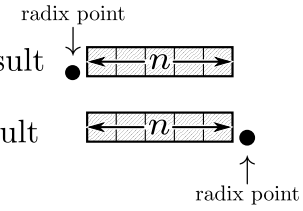
$$\Delta\mathcal{S}_i \rightarrow s_i 2^{m(a-i)}$$

which can also be written as:

$$\Delta\mathcal{S}_i \rightarrow s_i r^{(a-i)}, \quad r \stackrel{\text{def}}{=} 2^m$$

where s_i is the i 'th m -bit block, or the i 'th radix- 2^m digit of \mathcal{S}_n (Fig 2.3).

Note that depending on the position of the result's most-significant digit, the digit weights will vary, which is the purpose of the a parameter:

$$a = \begin{cases} 0 & \text{for a fractional result} \\ n & \text{for an integral result} \end{cases}$$


The diagram shows two horizontal bars, each divided into four segments and labeled with a right-pointing arrow and the letter 'n'. In the top case, labeled '0 for a fractional result', a dot labeled 'radix point' with a downward arrow is at the left end of the bar. In the bottom case, labeled 'n for an integral result', a dot labeled 'radix point' with an upward arrow is at the right end of the bar.

This way, there is no ambiguity as to what final precision to expect upon the end of the n 'th iteration. In this formulation, exactly n radix- 2^m digits, or $n \times m$ bits of \mathcal{S}_n are to be produced by the algorithm upon its completion.

2.3.3 Resulting algorithm

$$\begin{aligned} \mathcal{S}_0 &= 0 \\ R_0 &= N \\ 1 \leq i \leq n : \quad \mathcal{S}_i &= \mathcal{S}_{i-1} + s_i r^{a-i} \\ R_i &= R_{i-1} - 2\mathcal{S}_{i-1} s_i r^{a-i} - s_i^2 r^{2(a-i)} \end{aligned} \tag{2.3}$$

by future iterations of the algorithm. Each one of these digits is assigned an arbitrary range from $-\alpha$ (also written as $\bar{\alpha}$) to β , where h^+ and $-h^-$, in order, are the maximum- and the minimum-valued endless fractions that can be expressed using this digit set:⁽⁴⁾

$$h^- = \frac{\alpha}{r-1}, \quad h^+ = \frac{\beta}{r-1} \quad (2.4)$$

In the case of a conventional representation (which is of little practical interest), α would be equal to zero whereas β would be equal to $r-1$, implying a value for h^+/h^- of one and zero, respectively.

Note that unless either h^+ or h^- had a value of zero, a “less than” ($<$) sign should be used on both sides of the inequality above, which is typically the case in practical implementations of this algorithm and its derivative:

$$\mathcal{S}_{i-1} + s_i r^{a-i} - r^{a-i} h^- < S < \mathcal{S}_{i-1} + s_i r^{a-i} + r^{a-i} h^+$$

This inequality can be manipulated as follows:

$$\begin{aligned} \mathcal{S}_{i-1} + r^{a-i}(s_i - h^-) &< S < \mathcal{S}_{i-1} + r^{a-i}(s_i + h^+) \\ (\mathcal{S}_{i-1} + r^{a-i}(s_i - h^-))^2 &< \underbrace{S^2}_{=N} < (\mathcal{S}_{i-1} + r^{a-i}(s_i + h^+))^2 \\ \mathcal{S}_{i-1}^2 + 2\mathcal{S}_{i-1}r^{a-i}(s_i - h^-) &< N < \mathcal{S}_{i-1}^2 + 2\mathcal{S}_{i-1}r^{a-i}(s_i + h^+) \\ + r^{2(a-i)}(s_i - h^-)^2 && + r^{2(a-i)}(s_i + h^+)^2 \end{aligned}$$

At this point, we can simply transfer the term \mathcal{S}_{i-1}^2 to the middle to get $N - \mathcal{S}_{i-1}^2$, which is equal to the current remainder R_{i-1} :

$$\begin{aligned} 2\mathcal{S}_{i-1}r^{a-i}(s_i - h^-) &< R_{i-1} < 2\mathcal{S}_{i-1}r^{a-i}(s_i + h^+) \\ + r^{2(a-i)}(s_i - h^-)^2 && + r^{2(a-i)}(s_i + h^+)^2 \end{aligned} \quad (2.5)$$

⁽⁴⁾ These are often referred to as the *redundancy factors* in textbooks, which is a term avoided here. A more appropriate name of this amount would be the *positive* and the *negative fractional span*, respectively.

This inequality is called the *correctness constraint* and works as follows: depending on the value of the nominated digit s_i , it sets clear bounds on the current remainder R_{i-1} to determine whether that digit choice is a correct one or not.

Furthermore, the fact that the constraint revolves around the value of the current remainder R_{i-1} (rather than the value of the next remainder R_i) means that it can be used as the basis of a mechanism that permits the direct choice of digits, which is the goal of all digit-recurrence algorithms.

2.3.5 Paper-and-pencil method

A close cousin of the algorithm described above is a manual method for performing the square root, which is the paper-and-pencil square root algorithm. Though not taught in schools very often (compared to paper-and-pencil division), paper-and-pencil square root provides a good insight into the inner workings of the digit-recurrence square root, and hence is a valuable addition to this section.

Furthermore, the exact same technique used in the paper-and-pencil algorithm to combine the linear and the quadratic terms is also used in practical implementations of the digit-recurrence square root, adding to the relevance of this material.

Simplified correctness constraint

In paper-and-pencil square root, a conventional representation of \mathcal{S}_n is implicitly assumed ($\alpha = 0$, $\beta = r - 1$), with a radix commonly (but not necessarily) equal to ten.

Being a manual method (i.e. a method intended to be used by humans), *trial and error* is accepted as the principal mean of obtaining a correct choice of the result digits. This implies that unlike the correctness constraint in (2.5), which revolves around the value of the current remainder R_{i-1} (permitting direct selection of digits), a much simpler constraint that revolves around the value of the updated remainder R_i can be derived

for the paper-and-pencil method:

$$\begin{array}{ccc}
\boxed{\mathcal{S}_i \quad 0 \quad 0 \quad 0 \quad 0 \quad \dots} & & \boxed{\mathcal{S}_i \quad r-1 \quad r-1 \quad r-1 \quad r-1 \quad \dots} \\
\mathcal{S}_i \leq S < \mathcal{S}_i + r^{a-i} & & \\
\mathcal{S}_i^2 \leq S^2 < (\mathcal{S}_i + r^{a-i})^2 & & \\
\mathcal{S}_i^2 \leq N < \mathcal{S}_i^2 + 2\mathcal{S}_i r^{a-i} + r^{2(a-i)} & & \\
0 \leq R_i < 2\mathcal{S}_i r^{a-i} + r^{2(a-i)} & & (2.6)
\end{array}$$

Now since we know that one of the r digit choices is necessarily correct (due to the mathematical continuity of the square root function), only the lower bound of the upper constraint can be used to determine the value of the next result digit, by attempting to find the highest digit value s_i that does not result in a negative remainder.

Remember that the remainder R_i contains what remains of the radicand $N = S^2$ after extracting the square of the current result \mathcal{S}^2 from it, hence, the attempt to increase the value of the next digit choice without resulting in a negative remainder can be understood as trying to find the highest digit value that does not result in the new result \mathcal{S}_i exceeding the true square root S (through making sure that the square of the first does not exceed the square of the second).

Combining the linear and the quadratic terms

One complaint about the digit-selection tactic above (in case the same equations for the long square root were to be used) is the depth of the *trial-and-error cycle*. Namely, the user has to repeatedly subtract two terms, a linear and a quadratic term, just to see how this would affect the value of the next remainder R_i :

$$R_i = R_{i-1} - \underbrace{2\mathcal{S}_{i-1}s_i r^{a-i}}_{\text{linear term}} - \underbrace{s_i^2 r^{2(a-i)}}_{\text{quadratic term}} \quad (2.7)$$

To overcome this, we propose the *bare result* $\overline{\mathcal{S}}_{i-1}$ as the bare integer that results from combining all the result digits determined so far in the algorithm (Fig. 2.5).

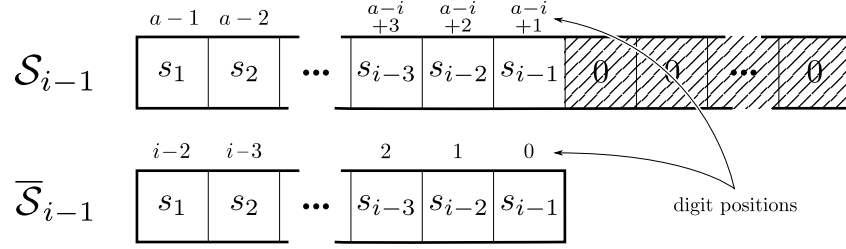


Figure 2.5: An illustration of the relationship between the result \mathcal{S}_{i-1} and the *bare result* $\overline{\mathcal{S}}_{i-1}$.

The new quantity, $\overline{\mathcal{S}}_{i-1}$, can be obtained from the current result \mathcal{S}_{i-1} by dividing by the weight of the least-significant, determined digit (this is the digit at the position number $\overset{a-i}{+1}$ in the figure):

$$\begin{aligned} \overline{\mathcal{S}}_{i-1} &\stackrel{def}{=} \mathcal{S}_{i-1} r^{-(a-i+1)} \\ \Rightarrow \quad &\boxed{\mathcal{S}_{i-1} \leftarrow \overline{\mathcal{S}}_{i-1} r^{a-i+1}} \end{aligned} \quad (2.8)$$

By substituting this into (2.7) we get:

$$\begin{aligned} R_i &= R_{i-1} - \underbrace{2\overline{\mathcal{S}}_{i-1} s_i r^{2(a-i)+1}}_{\text{linear term}} - \underbrace{s_i^2 r^{2(a-i)}}_{\text{quadratic term}} \\ &= R_{i-1} - s_i r^{2(a-i)} (2\overline{\mathcal{S}}_{i-1} r + s_i) \end{aligned}$$

Looking at the part within parentheses, it can be practically formed by concatenating $2\overline{\mathcal{S}}_{i-1}$ and s_i :

$$\begin{aligned} &= R_{i-1} - s_i r^{2(a-i)} \left(\boxed{2\overline{\mathcal{S}}_{i-1} | s_i} \right) \\ R_i &= R_{i-1} - r^{2(a-i)} \left(s_i \times \boxed{2\overline{\mathcal{S}}_{i-1} | s_i} \right) \end{aligned} \quad (2.9)$$

In this new formula, the current remainder R_{i-1} is updated through the subtraction of a single term (called the *linear-quadratic term*). This single term represents the change in the square of the current result \mathcal{S}_{i-1} due to the contribution of the last square root digit $s_i r^{a-i}$.

Note that the new term is fixed at two times the position of the last square root digit

(“ $2(a-i)$ ” vs. “ $a-i$ ”). This means that if we aggregate every couple of digits of the current remainder R_{i-1} , starting from the decimal point and moving outward, then the right edge of the linear-quadratic term would be neatly aligned with s_i (Fig. 2.6).

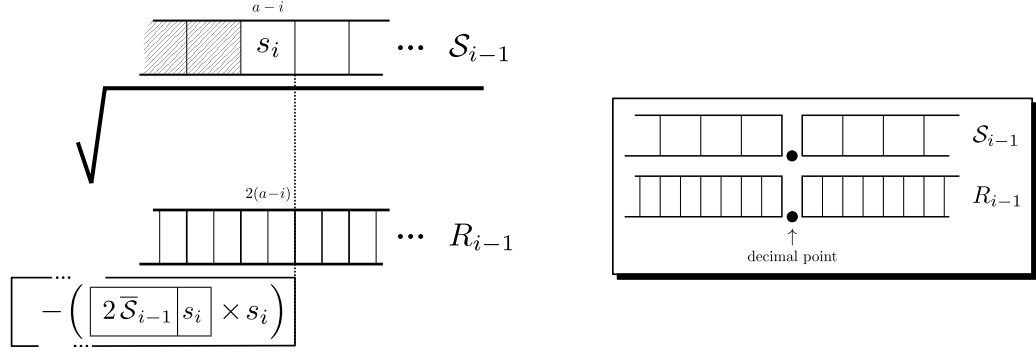


Figure 2.6: A diagram showing the correct relative position of the linear-quadratic term along the length of the current remainder R_{i-1} . Note that every digit of the square root is paralleled with a pair of remainder digits, and that the aggregation of remainder digits should always begin at the decimal/radix point and proceed outward, to meet the $a-i/2(a-i)$ digit-position relationship.

Combined with the simplified digit selection scheme in (2.6), the new formula corresponds to searching for the highest digit value “ $*$ ” such that $\left(* \times \boxed{2\overline{S}[*]} \right)$ is less than or equal to the value of the *relevant section* of R_{i-1} .

Decimal examples

	$\begin{array}{r} 1 \quad 2 \quad \leftarrow \text{result} \\ \sqrt{\underbrace{1 \quad 44}_{\bullet} \quad \leftarrow \text{radicand}} \end{array}$
$* \times \boxed{1 *}$	$- \frac{01}{00} \quad (* \rightarrow 1)$
	44
$* \times \boxed{2 *}$	$- \frac{44}{00} \quad (* \rightarrow 2)$

$$\begin{array}{r}
\begin{array}{c} 9 \ 8 \ 7 \\ \sqrt{97 \ 41 \ 69} \bullet \end{array} \\
* \times \boxed{1} * \quad - \frac{81}{16} \quad (* \rightarrow 9) \\
\begin{array}{c} 16 \ 41 \\ * \times \boxed{18} * \quad - \frac{15 \ 04}{1 \ 37} \quad (* \rightarrow 8) \\
\begin{array}{c} 1 \ 37 \ 69 \\ * \times \boxed{196} * \quad - \frac{1 \ 37 \ 69}{0 \ 00 \ 00} \quad (* \rightarrow 7) \end{array}
\end{array}$$

$$\begin{array}{r}
\begin{array}{c} 1 \bullet \ 4 \ 1 \ 4 \ 2 \ \dots \\ \sqrt{2 \bullet \ 00 \ 00 \ 00 \ 00 \ \dots} \end{array} \\
* \times \boxed{1} * \quad - \frac{01}{01} \quad (* \rightarrow 1) \\
\begin{array}{c} 1 \ 00 \\ * \times \boxed{2} * \quad - \frac{96}{04} \quad (* \rightarrow 4) \\
\begin{array}{c} 4 \ 00 \\ * \times \boxed{28} * \quad - \frac{2 \ 81}{1 \ 19} \quad (* \rightarrow 1) \\
\begin{array}{c} 1 \ 19 \ 00 \\ * \times \boxed{282} * \quad - \frac{1 \ 12 \ 96}{6 \ 04} \quad (* \rightarrow 4) \\
\begin{array}{c} 6 \ 04 \ 00 \\ * \times \boxed{2828} * \quad - \frac{5 \ 65 \ 64}{38 \ 36} \quad (* \rightarrow 2) \end{array}
\end{array}
\end{array}$$

2.3.6 Putting it all together

The same technique used in the above to combine the linear and the quadratic terms can also be incorporated into the long square root algorithm. Additionally, due to the binary nature of the long square root (as compared to the predominantly-decimal nature

of the paper-and-pencil method), further simplification is possible.

Namely, the $2\overline{\mathcal{S}}_{i-1}$ part of the linear-quadratic term can be further decomposed into the simple concatenation of the bare result word $\overline{\mathcal{S}}_{i-1}$ with a single zero bit $\boxed{\overline{\mathcal{S}}_{i-1}|0}$:

$$\begin{aligned}
\mathcal{S}_0 &= 0 \\
R_0 &= N \\
1 \leq i \leq n : \quad \mathcal{S}_i &= \mathcal{S}_{i-1} + s_i r^{a-i} \\
R_i &= R_{i-1} - r^{2(a-i)} \left(s_i \times \boxed{\overline{\mathcal{S}}_{i-1} | 0 | s_i} \right)
\end{aligned} \tag{2.10}$$

Result digits s_i are chosen from $\{-\alpha, \dots, \beta\}$ such that:

$$\begin{aligned}
2\mathcal{S}_{i-1}r^{a-i}(s_i - h^-) \\
+ r^{2(a-i)}(s_i - h^-)^2 &< R_{i-1} < 2\mathcal{S}_{i-1}r^{a-i}(s_i + h^+) \\
+ r^{2(a-i)}(s_i + h^+)^2
\end{aligned}$$

2.4 Incorporating Multiplication

In this section, we will take a brief diversion from the classical digit-recurrence theory to work up our novel inclusion of multiplication into the digit-recurrence square root.

This comes to complement, as well as to extend the work on building multiplication into the digit-recurrence division in [2], which is explored rapidly in the appendix.

Note that throughout this section, we will admittedly try to recreate the approach in [2], which will not only ease comparing the results, but will also increase the potential of either design by enabling the sharing of components between the two units (more on this in section 3.3 in Chapter 3).

2.4.1 An approximate radicand \tilde{N}_i

So far, the assumption has been that the radicand N is an exact amount that is provided in full precision at the beginning of the calculation:

$$N = S^2$$

This assumption has permitted us to define the remainder R_i as the (exact) error in the square, which is an amount of central importance to the current long square root algorithm (2.10):

$$\begin{aligned} R_i &\stackrel{def}{=} S^2 - \mathcal{S}_i^2 \\ &= N - \mathcal{S}_i^2 \end{aligned}$$

To define a fused operation, on the other hand, we're expected to carry out two mathematical operations at the same time, with the outcome of the first (in this case: multiplication) serving as the input of the second (in this case: the long square root).

To accomplish this, we're challenged to modify the long square root to accommodate an approximate radicand (i.e. a radicand that is still *under calculation*).

This starts by modifying the definition of the remainder above from being the exact error in the square $S^2 - \mathcal{S}^2$ ($= N - \mathcal{S}^2$) to becoming an approximation:

$$\text{an approximate remainder} \rightarrow \left(\tilde{R}_i \right) \stackrel{def}{=} \tilde{N}_i - \mathcal{S}_i^2 \quad (2.11)$$

radicand is now an *approximate* amount

radicand now needs a *subscript* since it is updated in every iteration

According to the definition above, introducing some error into the radicand $\epsilon(\tilde{N}_i)$ will result in introducing the same amount of error into the value of the remainder \tilde{R}_i :

$$\epsilon(\tilde{R}_i) = \epsilon(\tilde{N}_i)$$

where:

$$-\underbrace{\epsilon_{min}(\tilde{N}_i)}_{\substack{\text{absolute} \\ \text{minimum error}}} < \epsilon(\tilde{N}_i) < \underbrace{\epsilon_{max}(\tilde{N}_i)}_{\substack{\text{absolute} \\ \text{maximum error}}}$$

The major consequence of this modification is that the exact remainder R_{i-1} will no longer be available to the algorithm, meaning that the correctness constraint has to be rewritten with bounds on the approximate remainder \tilde{R}_{i-1} instead:

$$\begin{aligned} lowerbound &< \underbrace{R_{i-1}}_{\substack{= \tilde{R}_{i-1} + \epsilon(\tilde{R}_{i-1}) \\ = \tilde{R}_{i-1} + \epsilon(\tilde{N}_{i-1})}} < upperbound \\ lowerbound - \epsilon(\tilde{N}_{i-1}) &< \tilde{R}_{i-1} < upperbound - \epsilon(\tilde{N}_{i-1}) \end{aligned}$$

To account for the worst case, we add the absolute minimum error $\epsilon_{min}(\tilde{N}_{i-1})$ to the lower bound, and subtract the maximum positive error $\epsilon_{max}(\tilde{N}_{i-1})$ from the upper bound. This way, we ensure that the exact current residual R_{i-1} , if ever known, will surely lie in-between *lowerbound* and *upperbound*:

$$lowerbound + \epsilon_{min}(\tilde{N}_{i-1}) < \tilde{R}_{i-1} < upperbound - \epsilon_{max}(\tilde{N}_{i-1}) \quad (2.12)$$

2.4.2 Precision requirements of \tilde{N}_i

To obtain a general guideline of how small/large the error in the approximate radicand $\epsilon(\tilde{N}_{i-1})$ can be without obstructing the operation of the long square root algorithm, we need to have a good understanding of how the scale of *lowerbound* and *upperbound* changes from iteration to iteration (2.10):

$$\begin{aligned} lowerbound &= 2\mathcal{S}_{i-1}r^{a-i}(s_i - h^-) + r^{2(a-i)}(s_i - h^-)^2 \\ upperbound &= \underbrace{2\mathcal{S}_{i-1}r^{a-i}(s_i + h^+)}_{\text{term I}} + \underbrace{r^{2(a-i)}(s_i + h^+)^2}_{\text{term II}} \end{aligned} \quad (2.13)$$

Both bounds are formed by the addition of two terms, which will be referred to in

this discussion as “term I” and “term II”.

To investigate the scale as a function of the iteration number i , we need to express either term as the multiplication of a simpler, fixed-scale segment with an i -based scaling factor. To isolate this so-called scaling factor, we expand the current result \mathcal{S}_{i-1} above to the sum of its constituent, weighted digits $s_j r^{a-j}$:

$$\mathcal{S}_{i-1} = \sum_{j=1}^{i-1} s_j r^{a-j}$$

Note that despite of the sum’s upper limit above being a function of the iteration number i , the scale of \mathcal{S}_{i-1} is due to the greatest component which is independent of i (the greatest component in the sum above is $s_1 r^{a-1}$).

By substituting this into (2.13) we get:

$$\begin{aligned} lowerbound &= 2 \left(\sum_{j=1}^{i-1} s_j r^{a-j} \right) r^{a-i} (s_i - h^-) + r^{2(a-i)} (s_i - h^-)^2 \\ upperbound &= 2 \left(\sum_{j=1}^{i-1} s_j r^{a-j} \right) r^{a-i} (s_i + h^+) + r^{2(a-i)} (s_i + h^+)^2 \end{aligned}$$

where both expressions can be rearranged to highlight the scaling factor as follows:

$$\begin{aligned} lowerbound &= 2 \left(\sum_{j=1}^{i-1} s_j r^{-j} \right) \underbrace{(s_i - h^-) r^{2a-i}}_{\text{scaling factor/I}} + \underbrace{(s_i - h^-)^2 r^{2a-2i}}_{\text{scaling factor/II}} \\ upperbound &= 2 \left(\sum_{j=1}^{i-1} s_j r^{-j} \right) \underbrace{(s_i + h^+) r^{2a-i}}_{\text{scaling factor/I}} + \underbrace{(s_i + h^+)^2 r^{2a-2i}}_{\text{scaling factor/II}} \end{aligned}$$

In both cases, the scale of term I is determined by r^{2a-i} whereas the scale of term II is determined by r^{2a-2i} .

By discarding the invariable part r^{2a} , we can see that term I’s most-significant digit position is decreased by one in every iteration (implied by r^{-i}) whereas term II’s most significant digit position is decremented by two in every iteration (implied by r^{-2i}).

This implies that in the long run (in later iterations), the contribution of term I to the value of *lowerbound* and *upperbound* dominates, with an effective most-significant-

digit-position shift rate of one-digit-per-iteration, for each one of the two bounds.

Conclusion: in order to result in a feasible fused algorithm, both error terms $\epsilon_{min}(\tilde{N}_{i-1})$ and $\epsilon_{max}(\tilde{N}_{i-1})$ in (2.12) should have:

1. a controllable initial scale, and
2. a decrease rate no less than one digit position per iteration.

2.4.3 The direct approach

Now we're at the point of advocating a "recipe" for gradually calculating the radicand as the multiplication of two numbers $A \times B$, in a way that meets the precision requirements of the previous subsection.

To do this, we express the exact radicand $N = A \times B$ as the sum of n partial products⁽⁵⁾:

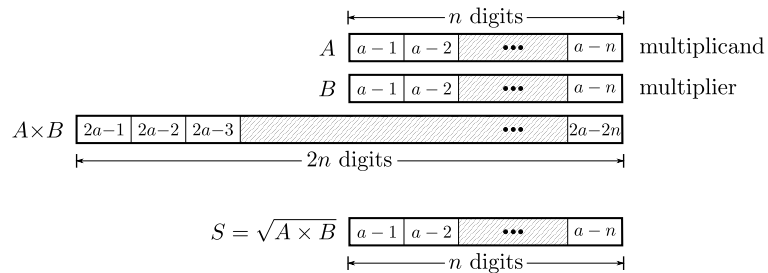
$$N = \sum_{j=1}^n b_j r^{a-j} A$$

As concluded in 2.4.2, this value has to be gradually calculated while (1) controlling the initial accuracy, and while (2) decreasing the error by one digit position per iteration.

A practical scheme that meets both requirements is the following:

1. one partial product is constantly added per iteration to the approximate radicand, following a strict most-to-least-significant ordering, while

⁽⁵⁾ The assumption here is that multiplying an n -digit multiplier B (whose digits are denoted as b_j), with an n -digit multiplicand A , will result in a $2n$ -digit radicand and an n -digit square root S , as follows:



2. the long square root algorithm is tuned to start at the δ 'th iteration, where δ is a system parameter that can be used to control the radicand's initial accuracy.

Note that a fused algorithm based on this scheme would require a total of $n + \delta - 1$ iterations to complete (compared to n iterations only for the long square root algorithm). This is due to the $\delta - 1$ *preliminary iterations* in which only partial products are being accumulated, but no result digits are being retired.

In this fused algorithm, the i 'th iteration will experience the determination of the $(i - \delta + 1)$ 'th result digit (in iterations starting from, and following the δ 'th iteration), compared to simply the i 'th result digit in a pure-square-root unit:

$$\mathcal{S}_i = \mathcal{S}_{i-1} + r^{a-(i-\delta+1)} s_{i-\delta+1} \quad (2.14)$$

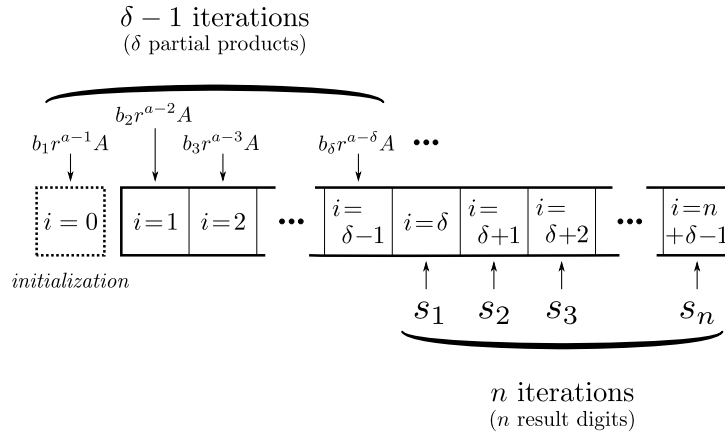


Figure 2.7

Initializing/updating the remainder \tilde{R}_i

In the development carried out in this subsection (and the following subsection), we choose to initialize the approximate radicand to the first partial product $b_1 r^{a-1} A$:

$$\tilde{N}_0 = b_1 r^{a-1} A$$

while adding one more partial product per iteration for the rest of the algorithm⁽⁶⁾:

$$\tilde{N}_i = \tilde{N}_{i-1} + b_{i+1}r^{a-i-1}A \quad (2.15)$$

This way, δ partial products will be accumulated prior to determining the first result digit (Fig. 2.7).

To derive an expression for updating the (now approximate) remainder \tilde{R}_i , we follow the same subtractive approach of subsection 2.2.2:

$$\tilde{R}_i = \tilde{R}_{i-1} + \Delta\tilde{R}_i$$

where:

$$\begin{aligned} \Delta\tilde{R}_i &\stackrel{\text{def}}{=} \tilde{R}_i - \tilde{R}_{i-1} \\ &= (\tilde{N}_i - \mathcal{S}_i^2) - (\tilde{N}_{i-1} - \mathcal{S}_{i-1}^2) \\ &= \underbrace{-(\mathcal{S}_i^2 - \mathcal{S}_{i-1}^2)}_{\substack{\text{evolves into the linear-quadratic-term } \mathcal{J} \\ \text{(same as before)}}} + \underbrace{(\tilde{N}_i - \tilde{N}_{i-1})}_{\substack{\text{to update the approximate radicand} \\ \text{(only in the fused algorithm)}}} \end{aligned}$$

This means that updating the approximate remainder will involve subtracting a linear-quadratic term similar to that of the long square root algorithm (2.10) as well as the addition of an extra, *partial product term*:

$$(\tilde{N}_i - \tilde{N}_{i-1}) = b_{i+1}r^{a-i-1}A$$

This, along with the delayed mapping between the result digit numbers and the iteration number (2.14), leads to the following full expression for updating the remainder:

$$\tilde{R}_i = \tilde{R}_{i-1} - \underbrace{r^{2(a-(i-\delta+1))} \left(s_{i-\delta+1} \times \boxed{\overline{\mathcal{S}_{i-1}} \mid 0 \mid s_{i-\delta+1}} \right)}_{\text{linear-quadratic term}} + \underbrace{b_{i+1}r^{a-i-1}A}_{\text{partial-product term}}$$

Note that in iterations preceding the δ 'th iteration, the value of the linear-quadratic

⁽⁶⁾ Note that upon using an n -digit multiplier B , which is the case assumed here, the last δ iterations of the $(n + \delta - 1)$ -iteration algorithm will be adding zero-valued partial products.

term will be equal to zero, resulting only in the additon of the partial product term in this initial phase of the fused algorithm.

As for the initial value of the approximate remainder \tilde{R}_0 , on the other hand, it is the same as the value of the initial approximate radicand⁽⁷⁾:

$$\begin{aligned}\tilde{R}_0 &\stackrel{def}{=} \tilde{N}_0 - \underbrace{\mathcal{S}_0^2}_{=0} \\ &= \tilde{N}_0 \\ &= b_1 r^{a-1} A\end{aligned}$$

where the approximate radicand \tilde{N}_i is initialized and updated as part of \tilde{R}_i , rather than being separately stored and maintained in the fused algorithm.

Putting the above together, we get the following multi-staged description for our fused multiplication/square-root algorithm:

$$\begin{array}{lcl} & \mathcal{S}_0 = 0 & \\ & \tilde{R}_0 = b_1 r^{a-1} A & \\ 1 \leq i \leq \delta - 1 & \left\{ \begin{array}{l} \tilde{R}_i = \tilde{R}_{i-1} + b_{i+1} r^{a-i-1} A \end{array} \right. & (2.16) \\ \delta \leq i \leq n + \delta - 1 & \left\{ \begin{array}{l} \mathcal{S}_i = \mathcal{S}_{i-1} + s_{i-\delta+1} r^{a-(i-\delta+1)} \\ \tilde{R}_i = \tilde{R}_{i-1} - r^{2(a-(i-\delta+1))} \left(s_{i-\delta+1} \times \boxed{\overline{\mathcal{S}_{i-1}} \mid 0 \mid s_{i-\delta+1}} \right) + b_{i+1} r^{a-i-1} A \end{array} \right. & \end{array}$$

Digit-selection criteria

Missing in the previous description is the digit-selection criteria, known previously as the correctness constraint, which is shown in (2.12) to have two extra terms compared to that of the long square root algorithm: $\epsilon_{min}(\tilde{N}_{i-1})$ and $\epsilon_{max}(\tilde{N}_{i-1})$ (called the absolute minimum and the absolute maximum errors in the radicand, respectively).

To find the value of either one of the two terms, we need to derive an analytical

⁽⁷⁾ Note that \mathcal{S}_0 in the fused algorithm is distinct from the initial root value, which is now equal to $\mathcal{S}_{\delta-1}$ (due to the δ -iteration delay in the operation of the square root).

expression for the error in the radicand $\epsilon(\tilde{N}_{i-1})$, as follows:

$$\begin{aligned}
& \boxed{
\begin{aligned}
N &= \sum_{j=1}^n b_j r^{a-j} A \\
\tilde{N}_{i-1} &= \sum_{j=1}^i b_j r^{a-j} A
\end{aligned}
} \\
\epsilon(\tilde{N}_{i-1}) &= N - \tilde{N}_{i-1} \\
&= \sum_{j=i+1}^n b_j r^{a-j} A \\
&= r^{a-i} \sum_{j=1}^{n-i} b_{i+j} r^{-j} A \\
\epsilon(\tilde{N}_{i-1}) &= r^{a-i} A \sum_{j=1}^{n-i} b_{i+j} r^{-j} \tag{2.17}
\end{aligned}$$

The absolute minimum and maximum values this expression can evaluate to depend on the ranges attainable by the multiplicand A , and the multiplier digits b_j .

Despite the fact that both A and B are provided in a conventional, irredundant digit representation, it is advantageous to assume that the digits of the multiplier b_j , at least, are converted to a signed-digit representation. This will reduce the number of A multiples that need to be generated within the algorithm to form the partial product term “ $b_{i+1} r^{a-i-1} A$ ” in (2.16).

Assuming a multiplier digit b_j in the range $\{B^-, \dots, B^+\}$, this results in the following, absolute-value limits of the sum part in (2.17):

$$\begin{aligned}
\left| \min \left(\sum_{j=1}^n b_j r^{-j} \right) \right| &< \left| \min \left(\sum_{j=1}^{\infty} b_j r^{-j} \right) \right| = h_B^- \quad \left(= \frac{B^-}{r-1} \right) \\
\left| \max \left(\sum_{j=1}^n b_j r^{-j} \right) \right| &< \left| \max \left(\sum_{j=1}^{\infty} b_j r^{-j} \right) \right| = h_B^+ \quad \left(= \frac{B^+}{r-1} \right)
\end{aligned}$$

where h_B^+ and $-h_B^-$, as before, are the maximum- and minimum-valued endless fractions that can be expressed using the signed digit set assigned to B .

Note that, should the multiplier-digit-conversion feature be dropped from the design, h_B^+ and h_B^- can simply be assigned the values of 1 and 0, respectively.

As for the multiplicand A , on the other hand, a conventional representation of digits is always assumed, leading to the following lower and upper limits:

$$\begin{aligned} \min(A) &= 0 \\ \max(A) &= r^a - r^{a-n} < r^a \end{aligned} \quad \begin{array}{c} \overbrace{\hspace{1.5cm}}^{n \text{ digits}} \\ A \quad \boxed{a-1 \mid a-2 \mid \cdots \mid a-n} \end{array}$$

By considering the limits above, we can see that to obtain an absolute-value minimum and maximum limits for (2.17), all we need to do is to pair the maximum multiplicand value $\max(A)$ with either the minimum, or the maximum value of the B -digit-sum part:

$$\begin{aligned} \epsilon_{\min}(\tilde{N}_{i-1}) &= r^{a-i} \times \max(A) \times \left| \min \left(\sum_{j=1}^n b_j r^{-j} \right) \right| < r^{2a-i} h_B^- \\ \epsilon_{\max}(\tilde{N}_{i-1}) &= r^{a-i} \times \max(A) \times \left| \max \left(\sum_{j=1}^n b_j r^{-j} \right) \right| < r^{2a-i} h_B^+ \end{aligned}$$

This implies that the error in the current radicand $\epsilon(\tilde{N}_{i-1})$ is bounded by:

$$\underbrace{-r^{2a-i} h_B^-}_{\epsilon_{\min}(\tilde{N}_{i-1})} < \epsilon(\tilde{N}_{i-1}) < \underbrace{r^{2a-i} h_B^+}_{\epsilon_{\max}(\tilde{N}_{i-1})}$$

which leads to the following, complete description of this subsection's fused algorithm:

$$\begin{aligned} \mathcal{S}_0 &= 0 \\ \tilde{R}_0 &= b_1 r^{a-1} A \\ 1 \leq i \leq \delta - 1 & \quad \left\{ \begin{array}{l} \tilde{R}_i = \tilde{R}_{i-1} + b_{i+1} r^{a-i-1} A \end{array} \right. \\ \delta \leq i \leq n + \delta - 1 & \quad \left\{ \begin{array}{l} \mathcal{S}_i = \mathcal{S}_{i-1} + s_{i-\delta+1} r^{a-(i-\delta+1)} \\ \tilde{R}_i = \tilde{R}_{i-1} - r^{2(a-(i-\delta+1))} \left(s_{i-\delta+1} \times \boxed{\overline{\mathcal{S}_{i-1}} \mid 0 \mid s_{i-\delta+1}} \right) + b_{i+1} r^{a-i-1} A \end{array} \right. \end{aligned} \tag{2.18}$$

Result digits $s_{i-\delta+1}$ are chosen from $\{-\alpha, \dots, \beta\}$ such that:

$$\begin{aligned} 2\mathcal{S}_{i-1} r^{a-(i-\delta+1)} (s_{i-\delta+1} - h^-) & \quad 2\mathcal{S}_{i-1} r^{a-(i-\delta+1)} (s_{i-\delta+1} + h^+) \\ + r^{2(a-(i-\delta+1))} (s_{i-\delta+1} - h^-)^2 & < \tilde{R}_{i-1} < + r^{2(a-(i-\delta+1))} (s_{i-\delta+1} + h^+)^2 \\ + r^{2a-i} h_B^- & \quad - r^{2a-i} h_B^+ \end{aligned}$$

2.4.4 An alternative approach

One inconvenience with the previous formulation in (2.18) is the distinction between iterations preceding and following the δ 'th iteration.

In a synchronous digital circuit, the same combinational hardware is reused between iterations. This requires, ideally, that the body of actions within the algorithm's loop be totally iteration-independent (where, in this case, the distinction between the iterations preceding and following δ can be seen as such dependence).

To resolve this, we propose an alternative scheme in which a fused multiplicative/square-root mechanical process is clung to throughout the algorithm, while effectively delaying the operation of the square-root part through deliberate attenuation of the radicand.

Hence, our goal will be to ensure that the $\delta - 1$ leading digits of the square root S/S be equal to zero (to recreate the scenario in Fig 2.7), which in turn, requires that we insert double this number of zero digits in the front of the radicand $A \times B$.

To do this in a manner that permits the accumulation of δ partial products prior to computing the square root digits, we need to incorporate all the $2(\delta - 1)$ leading zeros into the multiplicand A alone, rather than to the multiplier B , or by dividing this number of leading zero digits between the two amounts.

This technique can be demonstrated algebraically as follows:

$$\begin{aligned}
 S &= \sqrt{A \times B} \\
 r^{-(\delta-1)} S &= r^{-(\delta-1)} \sqrt{A \times B} \\
 \underbrace{r^{-(\delta-1)} S}_{\substack{S' \\ \text{(delayed)} \\ \text{result}}} &= \sqrt{\underbrace{r^{-2(\delta-1)} A \times B}_{\substack{A' \\ \text{(preprocessed)} \\ \text{multiplicand}}}} \\
 S' &= \sqrt{\underbrace{A' \times B}_{\substack{N' \\ \text{(processed)} \\ \text{radicand}}}}
 \end{aligned}$$

The computational equivalent of S' is \mathcal{S}' (with digits denoted as s'_i), which is the direct outcome of a single-staged, fused algorithm that utilizes this kind of downscaling

(or preprocessing) of the multiplicand:

$$A' = r^{-2(\delta-1)} A$$

To restore the actual result \mathcal{S}_{final} from the delayed one $\mathcal{S}'_{n+\delta-1}$, all we need to do is to drop the $\delta - 1$ leading zero digits:

$$\mathcal{S}_{final} = r^{\delta-1} \mathcal{S}'_{n+\delta-1}$$

Note that the approximate radicand \tilde{N}'_i is affected by the preprocessing of A in the same way the exact processed radicand N' is:

$$N' = r^{-2(\delta-1)} N$$

$$\tilde{N}'_i = r^{-2(\delta-1)} \tilde{N}_i$$

The approximate remainder is also affected, in almost the same way as the radicand:

$$\begin{aligned} \tilde{R}'_i &= \tilde{N}'_i - \mathcal{S}'_i{}^2 \\ &= r^{-2(\delta-1)} \tilde{N}_i - (r^{-(\delta-1)} \mathcal{S}_i)^2 \\ &= r^{-2(\delta-1)} \underbrace{(\tilde{N}_i - \mathcal{S}_i^2)}_{\tilde{R}_i} \\ \underbrace{\tilde{R}'_i}_{\substack{\text{processed} \\ \text{approximate} \\ \text{remainder}}} &= r^{-2(\delta-1)} \tilde{R}_i \end{aligned}$$

This effect on the remainder should to be taken into account in case the actual remainder, denoted as R_{final} , was to be recovered as one of the arithmetic unit's outcomes (in addition to the square root \mathcal{S}_{final} itself). In this case, the actual remainder is obtained by simply dropping the leading $2(\delta - 1)$ leading zero digits from the last remainder:

$$R_{final} = r^{2(\delta-1)} \tilde{R}'_{n+\delta-1}$$

This way, we can achieve a single-staged fused algorithm by replacing $A, \mathcal{S}_j, \overline{\mathcal{S}}_j, s_j, \tilde{N}_j$ and \tilde{R}_j with their processed equivalents: $A', \mathcal{S}'_j, \overline{\mathcal{S}}'_j, s'_j, \tilde{N}'_j$ and \tilde{R}'_j , respectively.

Note that the error in the radicand $\epsilon(\tilde{N}_{i-1})$ has also to be replaced by the error in the processed radicand $\epsilon(\tilde{N}'_{i-1})$, which will impact the error terms $\epsilon_{min/max}(\tilde{N}_{i-1})$ in (2.12):

$$\begin{aligned}
\epsilon_{min}(\tilde{N}'_{i-1}) &= \epsilon_{min}(r^{-2(\delta-1)}\tilde{N}_{i-1}) \\
&= r^{-2(\delta-1)}\epsilon_{min}(\tilde{N}_{i-1}) \\
&= r^{-2(\delta-1)}r^{2a-i}h_B^- \\
\epsilon_{max}(\tilde{N}'_{i-1}) &= \epsilon_{max}(r^{-2(\delta-1)}\tilde{N}_{i-1}) \\
&= r^{-2(\delta-1)}\epsilon_{max}(\tilde{N}_{i-1}) \\
&= r^{-2(\delta-1)}r^{2a-i}h_B^+
\end{aligned}$$

Putting all the above together, we get the following, enhanced fused algorithm:

$$\begin{aligned}
\mathcal{S}'_0 &= 0 \\
\tilde{R}'_0 &= b_1 r^{a-1} r^{-2(\delta-1)} A \\
1 \leq i \leq n + \delta - 1 \quad & \begin{cases} \mathcal{S}'_i = \mathcal{S}'_{i-1} + s'_i r^{a-i} \\ \tilde{R}'_i = \tilde{R}'_{i-1} - r^{2(a-i)} \left(s'_i \times \boxed{\overline{\mathcal{S}'_{i-1}} \mid 0 \mid s'_i} \right) + b_{i+1} r^{a-i-1} r^{-2(\delta-1)} A \end{cases} \\
\mathcal{S}'_{final} &= r^{\delta-1} \mathcal{S}'_{n+\delta-1} \\
R'_{final} &= r^{2(\delta-1)} \tilde{R}'_{n+\delta-1}
\end{aligned} \tag{2.19}$$

Result digits s'_i are chosen from $\{-\alpha, \dots, \beta\}$ such that:

$$\begin{aligned}
2\mathcal{S}'_{i-1}r^{a-i}(s'_i - h^-) & & 2\mathcal{S}'_{i-1}r^{a-i}(s'_i + h^+) \\
+ r^{2(a-i)}(s'_i - h^-)^2 & < \tilde{R}'_{i-1} < & + r^{2(a-i)}(s'_i + h^+)^2 \\
+ r^{-2(\delta-1)}r^{2a-i}h_B^- & & - r^{-2(\delta-1)}r^{2a-i}h_B^+
\end{aligned}$$

The description above looks and feels simpler than the one in (2.18) due to the unified loop's body, and due to the clean mapping between the digits of the delayed result s'_i and

the iteration number i , which hides the delay in the square root and the actual, shifted mapping between the two (Fig. 2.8).

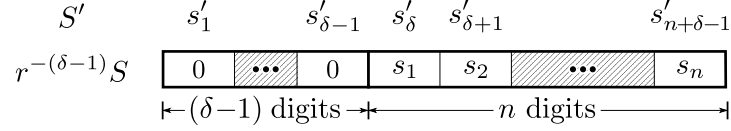


Figure 2.8: The mapping between the delayed square root digits s'_i and the digits of the actual square root $s_{i+\delta-1}$.

2.4.5 The Z parameter

A variation of the second approach in subsection 2.4.4 is to downscale A by a number of bits, rather than by whole digits (i.e blocks of m bits), as follows:

$$A' = 2^{-2Z} A$$

Preprocessing the multiplicand using binary shifts takes full advantage of the binary nature of modern digital systems, but introduces some complexities as well, which are explained in this subsection with the aid of the figure below.

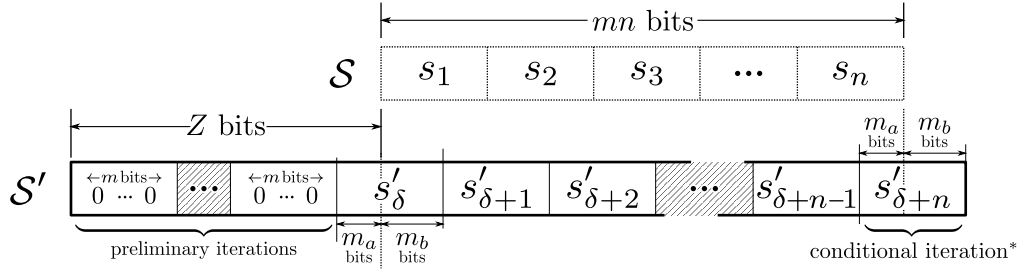


Figure 2.9

A case of special interest is the case when Z is not a multiple of m , that is to say, is the case when the Z parameter is used in a way that the δ parameter cannot match.

To address this case as well as the case when Z is simply a multiple of m , we redefine δ as a dependent variable where:

$$\delta \stackrel{\text{def}}{=} \left\lfloor \frac{Z}{m} \right\rfloor + 1 \quad (2.20)$$

By putting the ideas and definitions above into effect, we reach our final algorithm for this section, which will form the basis of the digit-recurrence algorithm of section 2.5.

$$\begin{aligned}
\mathcal{S}'_0 &= 0 \\
\tilde{R}'_0 &= b_1 r^{a-1} 2^{-2Z} A \\
1 \leq i \leq n^* + \delta - 1 & \begin{cases} \mathcal{S}'_i = \mathcal{S}'_{i-1} + s'_i r^{a-i} \\ \tilde{R}'_i = \tilde{R}'_{i-1} - r^{2(a-i)} \left(s'_i \times \boxed{\overline{\mathcal{S}'_{i-1}} | 0 | s'_i} \right) + b_{i+1} r^{a-i-1} 2^{-2Z} A \end{cases} \\
\mathcal{S}_{final} &= 2^Z \mathcal{S}'_{n^*+\delta-1} \\
R_{final} &= 2^{2Z} \tilde{R}'_{n^*+\delta-1}
\end{aligned} \tag{2.24}$$

Result digits s'_i are chosen from $\{-\alpha, \dots, \beta\}$ such that:

$$\begin{aligned}
2\mathcal{S}'_{i-1} r^{a-i} (s'_i - h^-) & \qquad \qquad \qquad 2\mathcal{S}'_{i-1} r^{a-i} (s'_i + h^+) \\
+ r^{2(a-i)} (s'_i - h^-)^2 & < \tilde{R}'_{i-1} < + r^{2(a-i)} (s'_i + h^+)^2 \\
+ 2^{-2Z} r^{2a-i} h_B^- & \qquad \qquad \qquad - 2^{-2Z} r^{2a-i} h_B^+
\end{aligned}$$

2.5 Finding a Recurrence Relation

In this section, we will continue our mission which we started in the previous section to remove all the iteration-dependent actions from the algorithm's loop in (2.24):

$$1 \leq i \leq n^* + \delta - 1 \quad \begin{cases} \mathcal{S}'_i = \mathcal{S}'_{i-1} + s'_i r^{a-i} \\ \tilde{R}'_i = \tilde{R}'_{i-1} - r^{2(a-i)} \left(s'_i \times \boxed{\overline{\mathcal{S}'_{i-1}} | 0 | s'_i} \right) + b_{i+1} r^{a-i-1} 2^{-2Z} A \end{cases}$$

This, as mentioned earlier, will enable reusing the same combinational hardware for executing each and every iteration, resulting in a significant reduction in the hardware complexity of the algorithm.

Note that as we try to do this, we are only allowed to change the internal products of the algorithm, as long as the operations (and hence the outcomes) are still the same.

By looking at the algorithm's loop above, we can see that updating the result \mathcal{S}' involves inserting the newly-elected digit s'_i at an iteration-dependent position (dictated by r^{a-i}), which is a simple example of an iteration-dependent action.

As for updating the remainder \tilde{R}'_i , on the other hand, the problem is a little bit more complicated as there are two iteration-dependent actions that are taking place in order to maintain it.

To help us identify and describe each one of the two actions, we view the linear-quadratic term and the partial-product term as the shifting of a simpler term, as follows:

$$\tilde{R}'_i = \tilde{R}'_{i-1} - \overbrace{r^{2(a-i)}}^{\text{shifting factor}} \underbrace{\left(s'_i \times \boxed{\overline{\mathcal{S}'_{i-1}} \mid 0 \mid s'_i} \right)}_{\substack{\text{linear-quadratic term} \\ \text{(simplified)}}} + \overbrace{r^{a-i-1} 2^{-2Z}}^{\text{shifting factor}} \underbrace{b_{i+1} A}_{\substack{\text{partial-product term} \\ \text{(simplified)}}} \quad (2.25)$$

Guided by this distinction, we can see that the lower end of either term is positioned at a different, yet iteration-dependent position (dictated by $r^{2(a-i)}$ and $r^{a-i-1} 2^{-2Z}$).

2.5.1 A recurrence relation for updating \mathcal{S}'

Speaking of \mathcal{S}' , we pay the reader's attention to the bare result $\overline{\mathcal{S}'_{i-1}}$ in the expression for updating the remainder, which is an amount introduced in subsection 2.3.5 on page 20 (in the part entitled *Combining the linear and the quadratic terms*).

As a reminder, the bare result (2.8) was defined as the “bare integer that results from combining all the result digits determined so far in the algorithm”, which is an amount easily formed digitally by feeding the newly-elected digits s'_i to a shift register⁽¹⁰⁾:

$$\overline{\mathcal{S}'_i} = r \overline{\mathcal{S}'_{i-1}} + s'_i \quad \overline{\mathcal{S}'} \quad \boxed{\text{shift register}} \quad \boxed{} \leftarrow s'_i$$

Not only that maintaining $\overline{\mathcal{S}'}$ instead of the actual result \mathcal{S}' will result in a simpler implementation, it will also supply this value for the update of the remainder, while not compromising the ease by which the final result \mathcal{S}_{final} is obtained.

⁽¹⁰⁾ Note that while this is true for an irredundant newly-elected digit s'_i (a one in the set $\{0, \dots, r-1\}$, comprising exactly m bits), this is not exactly the case with a redundant digit set of the form $\{-\alpha, \dots, \beta\}$. This will be one of the topics discussed in Chapter 4: *Hardware Design*.

To obtain the final result \mathcal{S}_{final} , we shift the final bare result $\overline{\mathcal{S}}'_{n^*+\delta-1}$ to the right by m_b bits to get rid of the excess bits (Fig. 2.9), then we reposition the lower end of this quantity at the final position, which is $a - n$, through multiplying by r^{a-n} :

$$\mathcal{S}_{final} = r^{a-n} 2^{m_b} \overline{\mathcal{S}}'_{n^*+\delta-1}$$

A nice property of the bare result is that it naturally discards the Z -bit-long preamble, which is evident in the expression above through the absence of any mention of Z .

2.5.2 A recurrence relation for updating \tilde{R}'

Now we get to the difficult part of the quest of this section, which is to find a recurrent (i.e. an iteration-independent) way of implementing (2.25).

The problem of updating the remainder in (2.25) is typical of the digit-recurrence algorithms, where to avoid subtracting a quantity from an iteration-dependent position, we change the location of the binary point between iterations, while matching the pace of the subtracted term, resulting in a fixed relative position (Fig. 2.10).

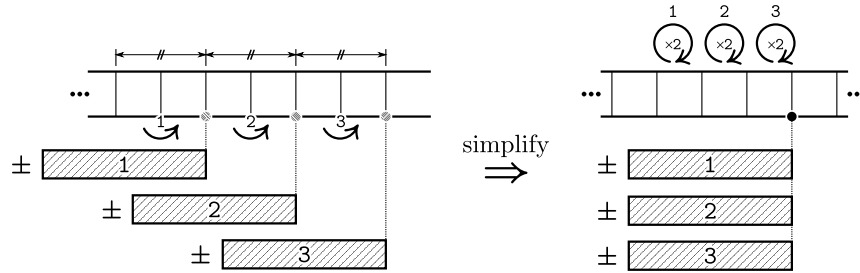


Figure 2.10: A scenario in which the lower end of the added/subtracted term is changed by two digit positions per iteration. The situation can be simplified by applying a double left shift to the base amount per iteration, resulting in a fixed relative position of the added/subtracted term.

Hence, the basic technique is to substitute a scale-changing quantity for \tilde{R}'_i , while shifting the contents of the new variable between iterations. This will simplify the loop's body by making it iteration-independent.

Unfortunately, applying this technique to the fused algorithm is made cumbersome by the partial-product term, which has a different pace from that of the linear-quadratic term (which is the only term in a pure square-root algorithm).

Namely, the lower end of the partial-product term changes position by one digit per iteration, whereas the lower end of the linear-quadratic term changes position by two digits per iteration.

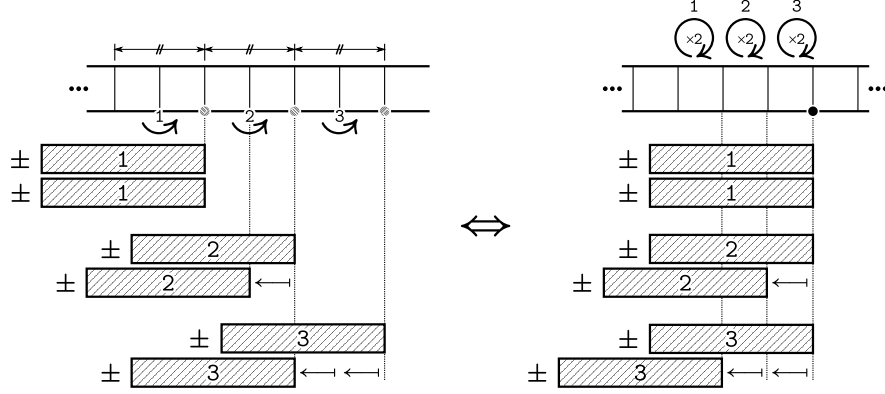


Figure 2.11: A modified scenario in which two terms are being added/subtracted per iteration, with the lower end of the first term changing position by two digits per iteration, and the lower end of the second term changing position by one digit per iteration. The situation can still be simplified by applying a double left shift per iteration, which will make the relative position of the first term fixed. As for the second term, on the other hand, we need to add/subtract it at an increasing lower-end position, to cancel one of the two left shifts applied to the base amount, thus meeting the required pace of one digit position per iteration.

To deal with this mismatch of pace between the two terms (Fig. 2.11), we propose the *shifted multiplicand* A_i^{\leftarrow} as a container-type variable whose function is to effectively cancel one of the two left shifts applied to the remainder in every iteration, to ensure that the actual multiplicand A within of A_i^{\leftarrow} will experience a single right shift per iteration:

$$\begin{aligned} A_0^{\leftarrow} &= A \\ A_i^{\leftarrow} &= r A_{i-1}^{\leftarrow} \end{aligned}$$

which implies that:

$$A_j^{\leftarrow} \stackrel{\text{def}}{=} r^j A \quad (2.26)$$

Note that as the case is with most of the other variables, the i 'th shifted multiplicand A_i^{\leftarrow} is the one computed, not consumed, in the i 'th iteration.

This means that to deploy this new variable in the algorithm's loop, we need to insert

the shifted multiplicand computed by the previous iteration A_{i-1}^\leftarrow :

$$\begin{aligned} A_{i-1}^\leftarrow &= r^{i-1}A \\ \Rightarrow \boxed{A \leftarrow r^{-i+1}A_{i-1}^\leftarrow} \end{aligned}$$

By substituting this into (2.25) we get:

$$\tilde{R}'_i = \tilde{R}'_{i-1} - r^{2a-2i} \left(s'_i \times \boxed{\overline{\mathcal{S}'_{i-1}} \mid 0 \mid s'_i} \right) + r^{a-2i} 2^{-2Z} b_{i+1} A_{i-1}^\leftarrow \quad (2.27)$$

Note that the shifting factor of either term has the same pace now (dictated by r^{-2i}), which is two digit positions per iteration.

This leads us to the point of applying the remainder-rescaling technique of Fig. 2.10.

To do this, we introduce the *residual* W_i as a variant of \tilde{R}'_i in which the binary point is moved to the right by two digit positions per iteration:

$$\begin{aligned} W_j &\stackrel{def}{=} r^{2j} \tilde{R}'_j \\ \Rightarrow \boxed{\tilde{R}'_i \leftarrow r^{-2i} W_i} \\ \Rightarrow \boxed{\tilde{R}'_{i-1} \leftarrow r^{-2i} W_{i-1} r^2} \end{aligned} \quad (2.28)$$

By substituting this into (2.27) we get:

$$r^{-2i} W_i = \leftarrow r^{-2i} W_{i-1} r^2 - r^{2a-2i} \left(s'_i \times \boxed{\overline{\mathcal{S}'_{i-1}} \mid 0 \mid s'_i} \right) + r^{a-2i} 2^{-2Z} b_{i+1} A_{i-1}^\leftarrow$$

where, by canceling the common r^{-2i} we get:

$$W_i = r^2 W_{i-1} - r^{2a} \left(s'_i \times \boxed{\overline{\mathcal{S}'_{i-1}} \mid 0 \mid s'_i} \right) + r^a 2^{-2Z} b_{i+1} A_{i-1}^\leftarrow \quad (2.29)$$

Note that according to the definitions in (2.26) and (2.28), both of A_0^\leftarrow and W_0 are to be assigned the same values as A and \tilde{R}'_0 , respectively.

As for restoring the actual remainder from the last residual, we substitute W for \tilde{R}'

in right side of the line for obtaining R_{final} in (2.24):

$$\begin{aligned}
 R_{final} &= 2^{2Z} \tilde{R}'_{n^*+\delta-1} & \boxed{\tilde{R}'_{n^*+\delta-1} \leftarrow r^{-2(n^*+\delta-1)} W_{n^*+\delta-1}} \\
 R_{final} &= r^{-2(n^*+\delta-1)} 2^{2Z} W_{n^*+\delta-1}
 \end{aligned}$$

2.5.3 Putting it all together

By putting the above together, we get the following, now named the *digit-recurrence multiplication/square-root algorithm*⁽¹¹⁾:

⁽¹¹⁾ Note that this algorithm is different from what is commonly known as the digit-recurrence algorithm due to the author's choice of moving the radix point of the residual by two digit positions per iteration. This deviation from classical theory is discussed at the end of Chapter 5: *Results and Discussion*.

$$\begin{aligned}
\overline{\mathcal{S}}'_0 &= 0 \\
W_0 &= b_1 r^{a-1} 2^{-2Z} A \\
A_0^\leftarrow &= A \\
1 \leq i \leq n^* + \delta - 1 & \begin{cases} \overline{\mathcal{S}}'_i = r \overline{\mathcal{S}}'_{i-1} + s'_i \\ W_i = r^2 W_{i-1} - r^{2a} \left(s'_i \times \boxed{\overline{\mathcal{S}}'_{i-1} \mid 0 \mid s'_i} \right) + r^a 2^{-2Z} b_{i+1} A_{i-1}^\leftarrow \\ A_i^\leftarrow = r A_{i-1}^\leftarrow \end{cases} \\
\mathcal{S}_{final} &= r^{a-n} 2^{m_b} \overline{\mathcal{S}}'_{n^*+\delta-1} \\
R_{final} &= r^{-2(n^*+\delta-1)} 2^{2Z} W_{n^*+\delta-1}
\end{aligned} \tag{2.30}$$

where:

m : number of result bits retired in every iteration

r : the radix (equals 2^m)

A : the *multiplicand*

$$A \quad \boxed{a-1} \mid \boxed{a-2} \mid \boxed{\dots} \mid \boxed{a-n}$$

b_j : j 'th digit of the *multiplier* B

$$B \quad \boxed{a-1} \mid \boxed{a-2} \mid \boxed{\dots} \mid \boxed{a-n}$$

Z : the number of zero bits in the preamble of \mathcal{S}'

n : number of m -bit digits in \mathcal{S}_{final}

n^* : number of fused-operation iterations (2.23)

δ : first iteration of the fused operation (2.20)

m_b : the *excess bits* in (2.22)

The algorithm revolves around maintaining three registers, to hold $\overline{\mathcal{S}}'_i$, W_i , and A_i^\leftarrow .

Chapter 3

SRT Table Design

In this chapter, we will cover the theory and techniques needed for obtaining the result digits s'_i in every iteration of the fused multiplication/square-root algorithm (2.30).

Furthermore, we will introduce the *Mathematica*[®]-based platform created as part of this research to explore the SRT-table design space, resulting in the system parameters leading to the most efficient design of the table. As will be shown later in this chapter, this platform implements a number of intriguing features that make searching for optimal designs exciting and fun.

The organization of this chapter is as follows: in section 3.1, we start by finding a recurrent form of the correctness constraint, which was intentionally omitted from the algorithmic description in (2.30), then, in section 3.2, we go over the body of ideas underlying the design of the SRT table (while highlighting features in the *Mathematica*[®]-based platform in asterisk-marked subsections), before we address the desire to share the SRT table with a fused multiplication/division unit in section 3.3. Then, in section 3.4, we discuss how to generate the actual contents of the SRT table, with a complete example for a maximally redundant, radix-4 system (featuring a shared table).

Last but not least, in section 3.5, we discuss the software organization and the design cycle dictated by our *Mathematica*[®]-based platform, while presenting some of the interesting system configurations that can be explored using this platform.

Note that throughout this chapter, subsections dedicated to highlighting features in

the software platform (possibly with code snippets and detailed explanations on how the code works), are marked with an asterisk to reflect the rather-practical nature of these subsections, and the fact that they might require some prerequisite knowledge in *Mathematica*[®] code, which is not otherwise assumed on the part of the reader.

3.1 A Recurrent Selection Criterion

In the digit-recurrence algorithms, m -bit fragments of the final result are sequentially revealed, which are used to update the indirect-error value stored in the remainder.

Each one of the m -bit fragments is called a digit, while the expression used to attune the remainder to the last digit choice is known as the *recurrence relation*.

The goal of digit selection is hence to find a value for the next result digit that keeps the remainder bounded, using *a priori knowledge* of the recurrence relation. This is done by putting bounds on the value of the current remainder, as follows:

$$lowerbound \left(\begin{array}{c} \text{input 2} \\ \boxed{\text{current} \\ \text{result}} \end{array}, \begin{array}{c} \text{output} \\ \boxed{\text{digit} \\ \text{choice}} \end{array} \right) < \begin{array}{c} \text{input 1} \\ \boxed{\text{current} \\ \text{remainder}} \end{array} < upperbound \left(\begin{array}{c} \text{input 2} \\ \boxed{\text{current} \\ \text{result}} \end{array}, \begin{array}{c} \text{output} \\ \boxed{\text{digit} \\ \text{choice}} \end{array} \right) \quad (3.1)$$

where the lower and the upper bounds are expressed as functions of the current result and the next digit choice, as shown above.

This implies that the digital device responsible for digit selection will take two inputs, one for the current remainder and one for the current result, while giving as output the result digit choice that meets the inequality.

Unlike novel attempts such as [18] (Chapter 4: *Comparison Multiples, a Different Approach to Quotient Digit Selection*), this device is prevalently implemented as a look-up table, commonly known as the *SRT table*.

To allow reusing the same SRT table between iterations, which is a core assumption of the digit-recurrence algorithms, the technical amounts advised for the inputs above

should lead to a converging lower and upper bounds⁽¹⁾:

$$\begin{aligned} \text{lowerbound} &= 2\mathcal{S}'_{i-1}r^{a-i}(s'_i - h^-) + r^{2a-2i}(s'_i - h^-)^2 + 2^{-2Z}r^{2a-i}h_B^- \\ \text{upperbound} &= \underbrace{2\mathcal{S}'_{i-1}r^{a-i}(s'_i + h^+)}_{r^{-i}} + \underbrace{r^{2a-2i}(s'_i + h^+)^2}_{r^{-2i}} - \underbrace{2^{-2Z}r^{2a-i}h_B^+}_{r^{-i}} \end{aligned}$$

Note that in the long run (in later iterations of the algorithm), the value of either bound will be mostly determined by the contribution of the first and the third terms (subsection 2.4.2), which are seen as the dominant part of the expression:

$$\begin{aligned} \text{lowerbound-dominant} &= 2\mathcal{S}'_{i-1}r^{a-i}(s'_i - h^-) + 2^{-2Z}r^{2a-i}h_B^- \\ \text{upperbound-dominant} &= 2\mathcal{S}'_{i-1}r^{a-i}(s'_i + h^+) - 2^{-2Z}r^{2a-i}h_B^+ \end{aligned} \tag{3.2}$$

To obtain an iteration-independent form of the dominant part above, we start by substituting the desired technical amount for the current result, then, through the choice of the right technical amount for the current remainder, we will make sure that the iteration dependence in the inequality in (3.1) will “mostly” get cancelled.

So let’s start by investigating the different technical amounts that can be substituted for \mathcal{S}'_{i-1} in (3.2), which will serve as one of the inputs to the SRT table.

The bare result $\overline{\mathcal{S}}'_{i-1}$, despite being a digitally-friendly amount that is directly maintained by the current algorithm (2.30), has the problem of having an unbounded range (the more digits get appended or shifted in to it, the greater the number becomes), and hence cannot be used to address the SRT table.

This leaves us with the more-conventional amounts that signify the result, with the choice between delayed, non-delayed, direct form, normalized-fraction form, etc.

To simplify the SRT-table design process, we opt for an amount that has a standard range, namely, a range that is independent from system parameters such as a or Z .

This suggests that we go for the non-delayed result \mathcal{S}_{i-1} (no prime) in a normalized-

⁽¹⁾ In this context, the word *converging* means “independent for the most part of i ”, which doesn’t imply that the expression be totally independent of i , only that the dominant part of it be.

fraction form $\dot{\mathcal{S}}_{i-1}$ (with a dot), which is defined as follows:

$$\dot{\mathcal{S}}_j \stackrel{def}{=} 2^Z r^{-a} \mathcal{S}'_j \quad (3.3)$$

where the multiplication factors 2^Z and r^{-a} serve the purpose of canceling the dependence on Z and a , respectively (Fig. 3.1).

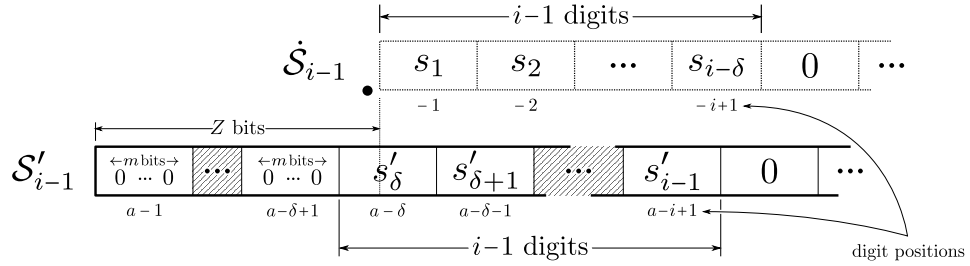


Figure 3.1: An illustration of the relationship between the digits of the delayed result \mathcal{S}'_{i-1} and the digits of the *fractional* result $\dot{\mathcal{S}}_{i-1}$.

To substitute the fractional result $\dot{\mathcal{S}}_{i-1}$ for \mathcal{S}'_{i-1} in (3.2), we apply:

$$\boxed{\mathcal{S}'_{i-1} \leftarrow 2^{-Z} r^a \dot{\mathcal{S}}_{i-1}}$$

to get:

$$\begin{aligned} \text{lowerbound-dominant} &= 2\dot{\mathcal{S}}_{i-1} r^{2a-i} 2^{-Z} (s'_i - h^-) + 2^{-2Z} r^{2a-i} h_B^- \\ \text{upperbound-dominant} &= \underbrace{2\dot{\mathcal{S}}_{i-1} r^{2a-i} 2^{-Z} (s'_i + h^+)}_{r^{-i}} - \underbrace{2^{-2Z} r^{2a-i} h_B^+}_{r^{-i}} \end{aligned}$$

where the dominant terms of either bound still have a pace of one digit position per iteration (dictated by r^{-i}).

This implies that a variant of the remainder in which the radix point is moved by one digit position per iteration is needed to mostly cancel the i -dependence in (3.1), leading to converging bounds that can be used as the basis of an all-iteration SRT table.

This is in contrast with the *residual* W directly maintained by the algorithm (2.30), which has a pace of two digit positions per iteration (2.28).

For this purpose, we introduce a new amount, which we call the *half-pace residual* Y :

$$\begin{aligned} Y_j &\stackrel{\text{def}}{=} r^j \tilde{R}'_j \\ \Rightarrow \boxed{\tilde{R}'_{i-1} \leftarrow r^{-i} Y_{i-1} r} \end{aligned} \quad (3.4)$$

By substituting the new amounts (Y_{i-1} and $\dot{\mathcal{S}}_{i-1}$) for \tilde{R}'_{i-1} and \mathcal{S}'_{i-1} in the correctness criterion on page 39, we get the following:

$$\begin{aligned} \overbrace{2\mathcal{S}'_{i-1} r^{a-i} (s'_i - h^-)}^{=2^{-Z} r^a \dot{\mathcal{S}}_{i-1}} &+ r^{2a-2i} (s'_i - h^-)^2 &< \underbrace{\tilde{R}'_{i-1}}_{=r^{-i} Y_{i-1} r} &< \overbrace{2\mathcal{S}'_{i-1} r^{a-i} (s'_i + h^+)}^{=2^{-Z} r^a \dot{\mathcal{S}}_{i-1}} \\ &+ 2^{-2Z} r^{2a-i} h_B^- &&+ r^{2a-2i} (s'_i + h^+)^2 \\ &&&- 2^{-2Z} r^{2a-i} h_B^+ \\ \\ 2\dot{\mathcal{S}}_{i-1} r^{2a} r^{-i} 2^{-Z} (s'_i - h^-) &&2\dot{\mathcal{S}}_{i-1} r^{2a} r^{-i} 2^{-Z} (s'_i + h^+) \\ &+ r^{2a-i} r^{-i} (s'_i - h^-)^2 &< r^{-i} Y_{i-1} r &< + r^{2a-i} r^{-i} (s'_i + h^+)^2 \\ &+ 2^{-2Z} r^{2a} r^{-i} h_B^- &&- 2^{-2Z} r^{2a} r^{-i} h_B^+ \end{aligned}$$

This inequality can be simplified by applying the following algebraic transformations: canceling the common r^{-i} , multiplying by 2^Z , and transferring 2^{2a} to the middle part.

This results in the following, recurrent form of the correctness criterion:

$$\begin{aligned} 2\dot{\mathcal{S}}_{i-1} (s'_i - h^-) + 2^{-Z} h_B^- &< \underbrace{r^{-2a} 2^Z (r Y_{i-1})}_{P_{i-1}} < 2\dot{\mathcal{S}}_{i-1} (s'_i + h^+) - 2^{-Z} h_B^+ \\ + 2^Z r^{-i} (s'_i - h^-)^2 &&+ 2^Z r^{-i} (s'_i + h^+)^2 \end{aligned} \quad (3.5)$$

where the lower and upper bounds are now mostly-independent of i , with a diminishing contribution of the lower term on both sides, which is the only part of this inequality that is iteration-dependent (due to the factor r^{-i}).

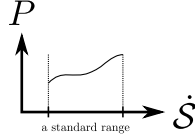
Note that the middle part can be rewritten as $P_{i-1}^{(2)}$, which is known as the *selection remainder*, for being the technical form of the remainder that is used for digit selection.

⁽²⁾ with P_j implicitly defined as $r^{-2a} 2^Z (r Y_j)$.

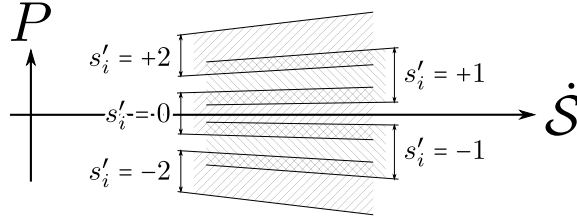
3.2 The P - \dot{S} Plane

An SRT table based on the criterion in (3.5) will have two inputs: \dot{S}_{i-1} and P_{i-1} , where the set of all the different combinations of the two inputs can be conveniently viewed as a 2-dimensional plane, referred to as the P - \dot{S} plane.

Since \dot{S} is designed to have a standard range, it is more intuitive to use the x -axis to convey the value of this input, rather than the other way around. This results in a diagram equivalent to the P - D diagram in the case of division, which we refer to as the P - \dot{S} diagram:



In the P - \dot{S} diagram, the vertical area enclosed between the upper and the lower *lines* of (3.5) is known as the selection region, where many such selection regions can be plotted for each possible choice of the next result digit $s'_i{}^{(3)}$:



3.2.1 Feasibility

As the value of P goes up and down, it passes from the area eligible for one of the digit choices to another, where, depending on the nature of that transition, three different feasibility categories can be defined.

Specifically, if P passed one of the selection regions to an area not eligible for any of the digit choices, which corresponds to the different selection regions having gaps in-between, then we are in front of an infeasible system in which the digit-selection task is impossible for some of the legitimate P/\dot{S} combinations.

⁽³⁾ Remember that $s'_i \in \{-\alpha, \dots, +\beta\}$.

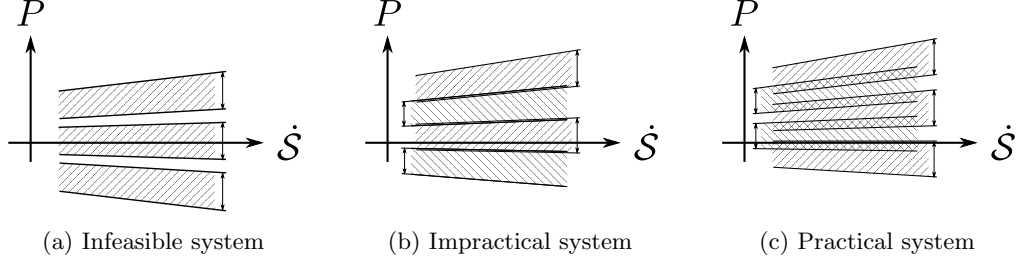


Figure 3.2: Three feasibility categories

If P , on the other hand, passed one of the selection regions to another, with little to no “overlap” in-between, then we are in front of a feasible, yet impractical system. This is because, even though there is always a valid choice for the next result digit, arriving at that choice might require the participation of all the bits of P and $\dot{\mathcal{S}}$, making the system unviable from a practical point of view.

This leaves us with the practical case where the selection regions succeed in covering the entire useable area of the plane while also providing a considerable overlap in-between, which corresponds to areas where more than one digit choice is valid. These so-called *overlap regions* will provide us with just enough tolerance to introduce some *uncertainty* in P and $\dot{\mathcal{S}}$, through the use of truncated samples (denoted as \hat{P} and $\hat{\mathcal{S}}$, respectively⁽⁴⁾).

A natural question at this point would be the question of how to change the category of the system or, the question of the system parameters that play a role in the system being feasible, impractical, or practical (which is the desired category), in Fig. 3.2.

This question translates to finding the parameters that increase the height of the selection regions and consequently, contribute to the existence or the height of the overlap regions, where increasing the height of the selection regions corresponds to increasing the value of the upper bound, decreasing the value of the lower bound, or both:

$$\underbrace{2\dot{\mathcal{S}}_{i-1}(s'_i - h^-) + 2^{-Z}h_B^- + 2^Z r^{-i}(s'_i - h^-)^2}_{\text{the lower bound}} < P_{i-1} < \underbrace{2\dot{\mathcal{S}}_{i-1}(s'_i + h^+) - 2^{-Z}h_B^+ + 2^Z r^{-i}(s'_i + h^+)^2}_{\text{the upper bound}} \quad (3.6)$$

⁽⁴⁾ Note that $\hat{\mathcal{S}}$ is used to refer to the truncated fractional result, instead of the more elaborate abbreviation as $\hat{\mathcal{S}}$. This is mainly to simplify the notation.

One factor that accomplishes both is the range of the result digit set $\{-\alpha, \dots, +\beta\}$, where a wider range (i.e. higher values for α and β) increases the value of h^- and h^+ , thus increasing the height of the selection regions through the double act of pushing the upper bound higher and the lower bound lower.

Another system parameter that plays a similar role is the Z parameter, which denotes the offset in the execution between multiplication and the square root.

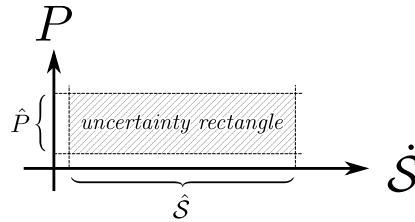
Namely, by opting for a more efficient integration of multiplication (a smaller value of Z), we are going to impact the height of the selection regions (and that of the overlap regions, thereof) negatively (through the terms $+2^{-Z}h_B^-$ and $-2^{-Z}h_B^+$)⁽⁵⁾.

3.2.2 Comparison constants

As mentioned earlier, practical systems use the overlap regions to absorb the amount of uncertainty associated with a truncated remainder \hat{P} and a truncated result \hat{S} , which are the inputs seen by the digit-selection device (i.e. the SRT table).

Note that \hat{P} and \hat{S} are obtained from P and \dot{S} by truncating the latter amounts to n_P and n_S fractional bits, respectively.

Unlike the non-truncated amounts, truncated inputs correspond to a range of (rather than only one) possible values, where a pair of truncated inputs maps to a rectangle in the P - \dot{S} diagram, occasionally known as the *uncertainty rectangle*:



Consequently, for there to be a valid digit-selection outcome associated with a pair of truncated inputs $\{\hat{P}, \hat{S}\}$, it's required that the corresponding uncertainty rectangle be entirely within the selection region associated with that outcome.

⁽⁵⁾ Hence, choosing the Z parameter is a compromise between the wish for a more efficient integration of multiplication with the base operation, while still providing enough overlap between the selection regions.

Furthermore, as the entire $P\text{-}\dot{S}$ plane is now navigated through truncated values of P and \dot{S} , this defines how the height of the overlap regions is to be determined, in terms of the region's *effective height* (Fig. 3.3).

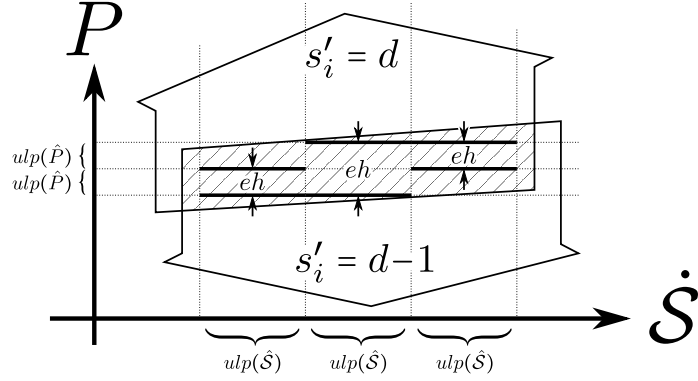


Figure 3.3: The *effective height* of the overlap region at different values of \dot{S} . Note that the steps $ulp(\hat{P})$ and $ulp(\hat{S})$ are equal to 2^{-n_P} and 2^{-n_S} , respectively.

This implies that the overlap region's effective height can be zero when there is in fact a bit of an overlap between the selection regions, furthermore, it implies that the overlap region might be effectively nonexistent, even when it actually exists (Fig. 3.4).

To alter the effective geometry of the overlap regions, that is, to make an effectively-nonexistent region exist or increase the effective height, the dotted lines above has to be brought closer by reducing the ulp of \hat{P} , \hat{S} , or both, which corresponds to truncating fewer bits of P or \dot{S} (i.e. a higher value for n_P or n_S).

The reason that makes the effective geometry of the overlap regions so important for the design of the SRT table is the fact that such areas mark the values of \hat{P} associated

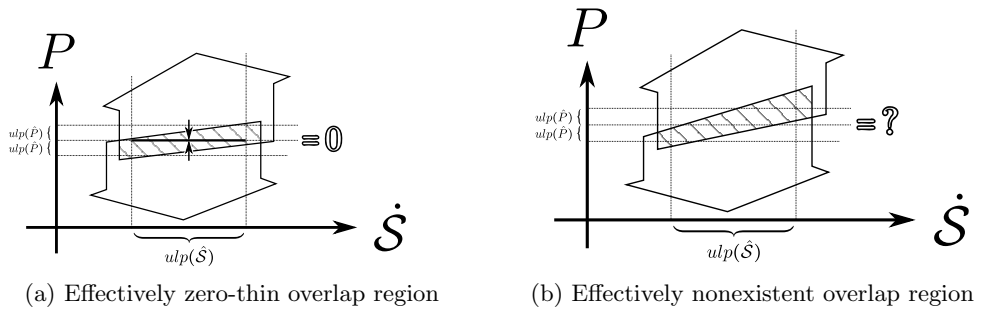


Figure 3.4: Effect of quantization on the geometry of the overlap regions.

with a transition from one of the digit choices to another, which are known as the *comparison constants*.

Namely, for each possible value of the truncated result $\hat{\mathcal{S}}$ there exists $\alpha + \beta$ comparison constants associated with the digit choices from $-\alpha + 1$ to $+\beta$, where the digit choice is based on the highest constant (for that value of $\hat{\mathcal{S}}$) that is exceeded or equalled by the value of \hat{P} (Fig. 3.5).

Note that if non of the comparison constants were exceeded or equalled by the value of \hat{P} , then the lowest digit choice $-\alpha$ is given as the output, which is why only $\alpha + \beta + 1$ constants are needed for making a choice between $\alpha + \beta + 1$ digit values.

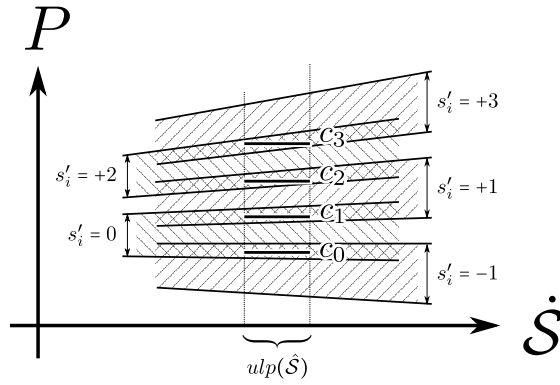


Figure 3.5: A depiction of the *comparison constants*. Note that 4 constants are needed to make a choice between 5 possible digit values from -1 to +3. Also note that a similar set of comparison constants would be needed for every possible value of the truncated result $\hat{\mathcal{S}}$.

COMPUTING THE EFFECTIVE GEOMETRY*

This is the first of a number of subsections that are dedicated to explaining how the *Mathematica*[®]-based platform developed as part of this research accomplishes some of the theoretical tasks outlined elsewhere.

In this subsection, we will explain how to compute the overlap region's effective geometry, which is done starting from the continuous selection region's lower/upper-bound function, named `delta` (3.6):

$$\text{delta} := 2X \times \text{digit} + 2^Z \text{radix}^{-\text{iteration}} \text{digit}^2 - 2^{-Z} B \text{redundancy_factor}$$

Note that the variable `X` above conveys the value of the current result \mathcal{S}'_{i-1} or in

the case of sharing the SRT table with division, can also be used to convey half the value of the divisor (more on this in section 3.3).

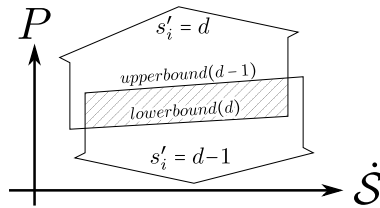
As for the variable `digit`, it will be used to convey either $(s'_i - h^-)$ or $(s'_i + h^+)$, depending on whether the substitution occurs in the lower or the upper bound of the selection region. In the same way, the variable `B`redundancy`factor` will be used to denote either h_B^- or h_B^+ , depending on the context.

Starting from this function, the effective geometry is computed in two steps: by putting the quantization of $\dot{\mathcal{S}}$ (X above) into effect first, then putting the quantization of P (represented by the value of the whole expression above) into effect second.

To do this, we create a list of the lower and upper lines enclosing the $\alpha + \beta$ overlap regions in the P - $\dot{\mathcal{S}}$ diagram:

```
overlap`lines=Table[{
  delta /. {digit→(i-h-),B`redundancy`factor→-hB-},
  delta /. {digit→((i-1)+h+),B`redundancy`factor→+hB+}
},{i,-α+1,+β}]
```

where, as you can see, the i 'th overlap region is enclosed between the lower bound of the i 'th selection region and the upper bound of the $i - 1$ 'th selection region.



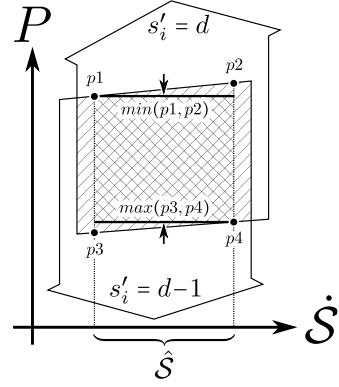
To move a step forward from this point, we put the quantization of X (the truncated result) into effect by duplicating this set of lines for every potential value of X within its range, while substituting the continuous upper and lower lines with a worst-case lower and upper values, dictated by the intervals associated with each one of the possible X values (a list of such intervals is stored in `xintervals` below):

```

intervals=Join@@Outer[
  (
    Max[
      #2[[1]]/.{iteration→#1[[2]],X→#1[[1,1]]},
      #2[[1]]/.{iteration→#1[[2]],X→#1[[1,2]]}
    ],
    Min[
      #2[[2]]/.{iteration→#1[[2]],X→#1[[1,1]]},
      #2[[2]]/.{iteration→#1[[2]],X→#1[[1,2]]}
    ]
  )&,

  MapThread[List,{xintervals,xintervals`iteration}],
  overlap`lines,
  1];

```



Note that not only this would result in an overlap region (now more properly called an *interval*) for every s'_i/\hat{S} combination, the set is also duplicated for all the iterations of the algorithm, since the iteration number plays a role in determining the value of the `delta` expression above (more on this in subsection 3.2.3).

From this point on, it is easy to put the quantization of P into effect by dividing the overlap-interval endpoints by the *ulp* of \hat{P} , which is 2^{-n_P} :

$$\text{intervals`gridsteps} = \text{intervals}/.\{\text{lower_}, \text{upper_}\} \rightarrow \left\{ \frac{\text{lower}}{2^{-n_P}}, \frac{\text{upper}}{2^{-n_P}} \right\};$$

then, flooring (\downarrow) the upper and ceiling (\uparrow) the lower endpoints of the intervals, to find the truncated values less than the upper, and greater than the lower grid-step values.

Note that if we used the *Mathematica*[®]-provided functions `Floor` and `Ceiling` for this purpose, then we might end up with an upper or a lower endpoint equal to those of the grid-step intervals above, which would violate the strictly-less-than ($<$) and the strictly-greater-than ($>$) operators of the correctness criterion in (3.6).

To deal with this delicate problem, we define our own *conservative* edition of both functions, as follows:


```

ConserCeiling[number_] := (Ceiling[number] + Boole[IntegerQ[number]]);
ConserFloor[number_] := (Floor[number] - Boole[IntegerQ[number]]);

```

where the original ceiling is conditionally incremented by one, and the original floor is conditionally decremented by one, to account for the case when the endpoint is already an exact multiple of $ulp(\hat{P})$.

Note that before these routines are applied to either endpoint of the intervals, the intervals are *intersected* to account for all the iterations of the algorithm, for custom interval mappings, or for a symmetric table (more on the topic of intersecting intervals in subsection 3.2.3).

3.2.3 Iteration dependence

In the asterisk-marked section `COMPUTING THE EFFECTIVE GEOMETRY*`, we mentioned that many overlap intervals are computed for the same next result digit s'_i and truncated result $\hat{\mathcal{S}}$ combination, for the different iterations of the algorithm, which is due to the iteration-dependent term $2^Z \text{radix}^{-\text{iteration}} \text{digit in delta}$.

As you might remember, we started this chapter by attempting to find a converging form of the lower and the upper bounds of the correctness constraint, which resulted in the current form of the `delta` expression.

It was mentioned that the attempt was not to make the expression iteration-independent, but to make the contribution of the iteration number i minor to the overall value of the expression, which was achieved by substituting a variant of the *half-pace residual* Y for the *selection remainder* P in the correctness constraint (page 50).

One property of the current contribution of i to the lower and the upper lines of the overlap regions is that it is of a diminishing nature (due to $\text{radix}^{-\text{iteration}}$), where only the first few iterations (called the *early iterations*) would be significantly affected by this contribution.

Note that in the fused algorithm, this corresponds to the few iterations starting from, and following the δ 'th iteration (page 37).

For this reason, designers have classically took the route of trying to provide the result

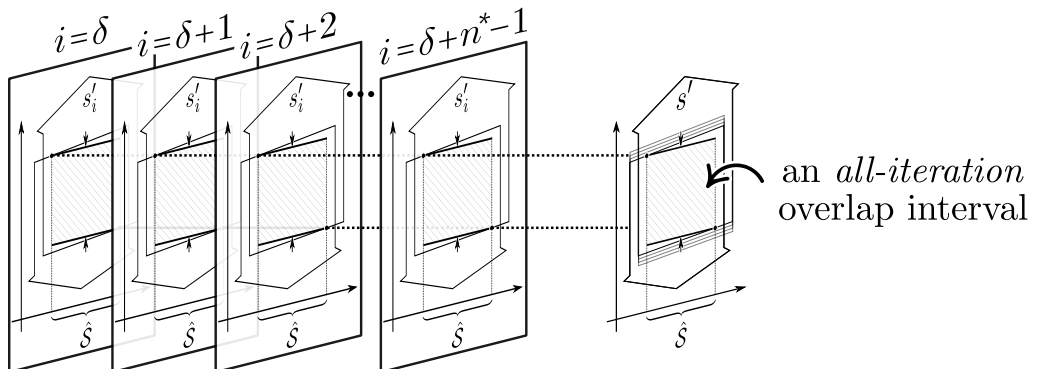
digits during these early iteration through other means, such as a separate look-up table (which is to be indexed by the few most-significant bits of the radicand). This look-up table is sometimes called the *initial PLA* [11] [13] [17] [27], which was occasionally escaped by opting for a smart initialization scheme [16], or through a combination of techniques—in the influential work in [8].

In this thesis, we will take the latter route of applying a number of techniques which, along with a special initialization scheme, will eliminate the need for an initial PLA⁽⁶⁾. In fact, many of the intriguing features in the *Mathematica*[®]-based platform of this thesis are mere extension of the powerful ideas present in [8].

INTERSECTING THE INTERVALS*

To merge all the overlap intervals associated with one s'_i/\hat{S} combination, in preparation to quantizing the interval, and choosing a value for the comparison constant associated with it, we apply a simple technique that we call *intersecting the intervals*.

This technique is illustrated in the figure below, where all the intervals associated with one s'_i/\hat{S} combination (called the **relevant intervals** in the code) are intersected by finding the maximum lower endpoint, and the minimum upper endpoint. This results in a worst-case interval that is valid for all iterations of the algorithm⁽⁷⁾.



⁽⁶⁾ Note that despite of this work not suggesting the use of an initial PLA, it does require a custom device called the *First-Digit Selector*, to deal with the extended \hat{S} range in the fused multiplication/square-root unit. This is the topic of section 4.2 on Chapter 4.

⁽⁷⁾ Note that despite of the iteration $i = \delta$ being included in the figure, the suggested architecture of this thesis will in fact advocate the use of a First-Digit Selector to provide the digit selection at this iteration, which implies that the selection intervals associated with the δ 'th iteration can be dropped from the SRT table design process.

Hence, if a comparison constant was selected from within this *intersected* interval, it will be valid for all iterations of the algorithm.

The dynamic *Mathematica*[®] line that performs this task is the following:

```
intersected`interval:=Which[
    relevant`intervals!={},
    {ConserCeiling[Max[(Transpose@relevant`intervals)[[1]]]],
    ConserFloor[Min[(Transpose@relevant`intervals)[[2]]]],True,{}};
```

Note that the line uses delayed assignment ($:=$), which is essential for the code interactive abilities (in case we forgot to mention, the platform has interactive functions where you can alter the intervals in some way, and watch the results instantly).

Note also the use of the *conservative floor* and the *conservative ceiling* functions, which were documented earlier in the asterisk-marked section in 3.2.2.

It's important to mention at this point that collapsing the intervals in this fashion is not enough to result in feasible intervals (i.e. intervals from which a valid comparison constant can be chosen). In fact, what is needed is a good plan for *excluding* some of the problematic (and unnecessary) intervals originating in the early iterations of the square root, which will be the subject of subsection 3.2.5.

Note that the technique of intersecting the intervals is a versatile one, which can be used not only to eliminate the iteration dependence, but also to design a symmetric table where negative values of \hat{P} are negated prior to using them to index the table, resulting in a half-sized table. Another utilization of this technique is the deployment of custom mappings, where both topics are discussed in subsections 3.2.6 and 3.2.7, respectively, and are considered advanced topics.

3.2.4 Determination of the constants

In this subsection, we will discuss the overlap region's effective-height requirements leading to the existence of all the $\alpha + \beta$ comparison constants (for each of the possible values of the truncated fractional result \hat{S}).

Note that if these minimum heights were not met by any of the overlap regions in the $P\text{-}\hat{S}$ diagram, then the comparison constant associated with that overlap region would

not exist, implying an infeasible SRT table.

To fix this problem, we have the choice between: (1) increasing the amount of overlap, by adding more digit values to the result's digits set⁽⁸⁾, or (2) enhancing the overlap regions' effective geometry through an increased value of n_P , n_S , or both.

Note that for any of the comparison constants to exist, it's required that:

1. The range of values associated with \hat{P} when equal to the constant be entirely within the selection region implied by it.
2. The range of values associated with \hat{P} when equal to the truncated value preceding the value of the constant be entirely within the lower selection region.

Hence, the question of the minimum overlap region's effective height is closely coupled with the question of the range of values (i.e. the *domain*) associated with the different values of \hat{P} , which, in turn, is a function of the internal representation of P (Fig. 3.6).

By constructing theoretical, minimal-height overlap regions for each one of the three representations in Fig. 3.6, we reach the minimum height requirement of 0, $ulp(\hat{P})$, and $ulp(\hat{P})$ for an *irredundant*⁽⁹⁾, *carry-save*, and a *signed-bit* representations (Fig. 3.7).

In a similar way, we can obtain the *codomain* for choosing the comparison constants, by constructing theoretical, maximal-height overlap regions for each one of the three cases. This results in the codomains of Fig. 3.8.

⁽⁸⁾ If in need, revise the last few paragraphs of subsection 3.2.1.

⁽⁹⁾ An irredundant format is nothing but the number represented in bits, as one binary word.

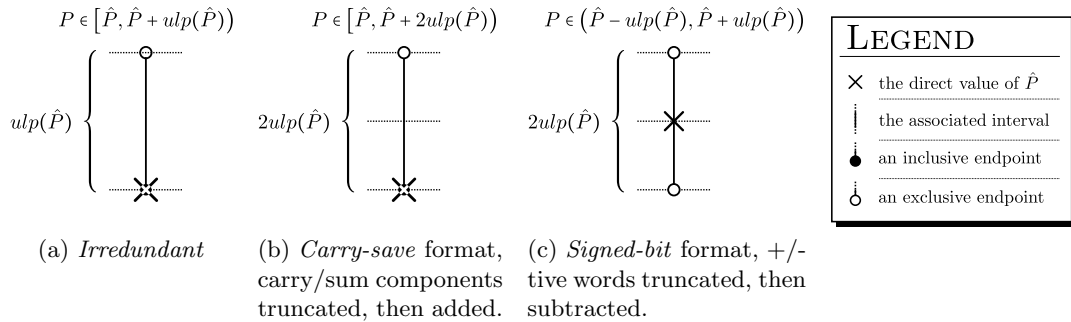


Figure 3.6: The effect of the internal representation of P on the truncation intervals. Note that in the case of the carry-save and the signed-bit representations, both components are truncated to n_P fractional bits before being added/subtracted.

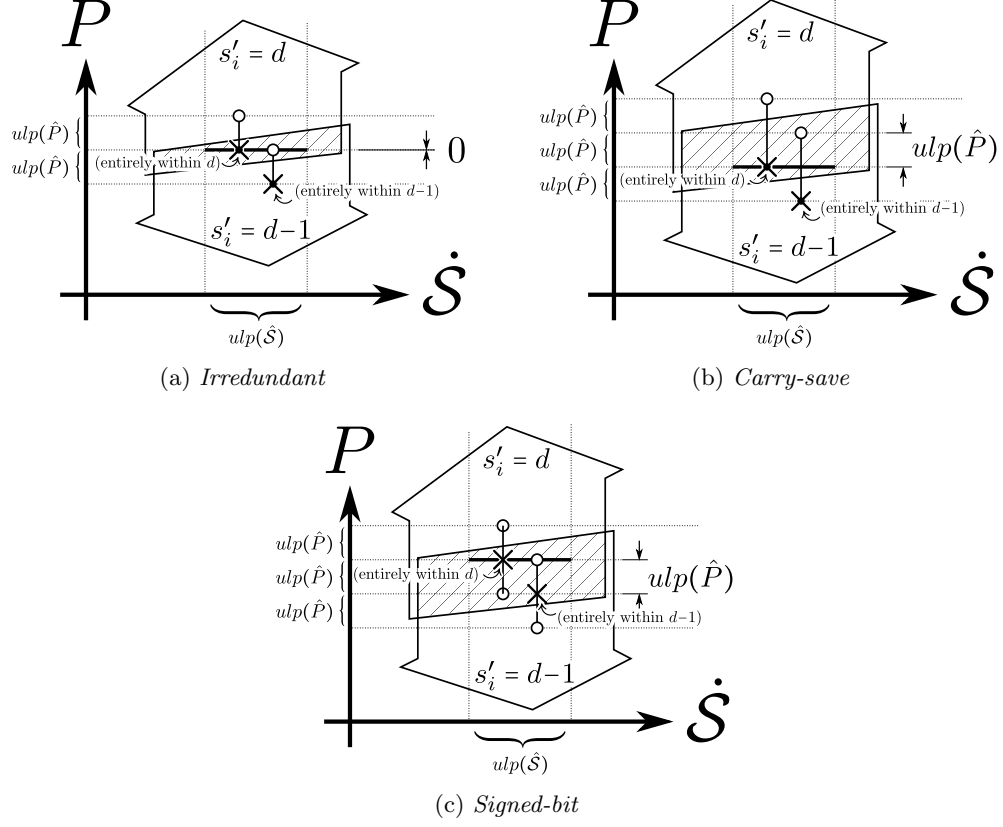


Figure 3.7: The minimum overlap region's effective height for different representations of P .

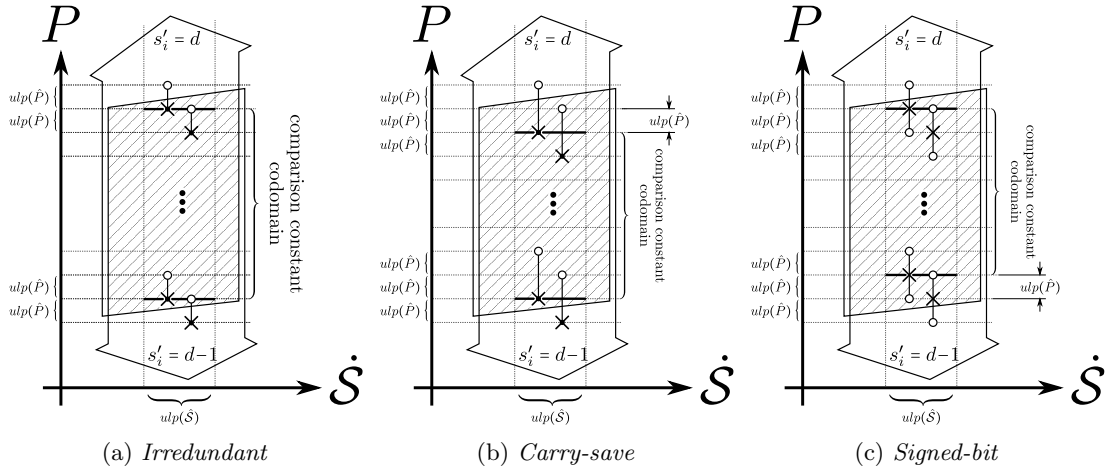


Figure 3.8: The target set for the comparison constants depending on the internal representation of P . Note that the uppermost value within the effective height is excluded in the case of a carry-save P , while the lowermost value is excluded in the case of a signed-bit one.

SMART CHOICE OF THE COMPARISON CONSTANTS*

Now that we have arrived at the target set for choosing the comparison constants, a legitimate question is whether there exists a more or a less judicious way of choosing the constants within their assigned codomain, and whether it would make a difference.

Our answer to this is the *smart choice* feature, which aims at finding the value of the comparison constant that features the least number of bits.

This is done by constructing a list of all the possible values of the constant within its codomain, converting them to a binary representation, and then sorting the candidates by bit length. After that, any of the shortest-length values is chosen, by assigning the value of the constant to the first value in the sorted list.

This is very easily done in *Mathematica*[®] script, as follows:

```
smart`choice[codomain_] :=  
  FromDigits[First[SortBy[RealDigits[  
    codomain, 2], (Length[#[[1]]] - #[[2]]&)], 2];
```

Note that the `RealDigits` function returns binary numbers in the following format:

$$\{\{\text{list of bits}\}, \text{number of integral bits}\}$$

So what the code above does is that it sorts a list of binary numbers (in the above format) using the criterion $(\text{Length}[\#[[1]]] - \#[[2]])$, which subtracts the number of integral bits (the second component of each converted number) from the total number of bits (the length of the first component, which is a list of bits). This results in the potential constant values being sorted in the order of increased fractional-bit count.

After that, the first element is picked up (using the `First` function) and converted back from binary to normal (using the `FromDigits` function).

To generate the codomain from a lower endpoint `codomain`lower`, an upper endpoint `codomain`upper`, and a value of $ulp(P)$ of 2^{-n_P} , we can use the `Range` function:

```
Range[codomain`lower, codomain`upper, 2-nP]
```

Example 1: Assuming a codomain of $\{\frac{57}{16}, \frac{67}{16}\}$ and a step (i.e. a P 's *ulp*) of $\frac{1}{16}$, we get the smart choice of 4., which has no fractional bits at all.

Example 2: Using a tighter codomain of $\{\frac{49}{16}, \frac{55}{16}\}$ and the same step, we can still get a good smart choice of $\frac{13}{4}$, which has only two fractional bits.

SMART-CONSTANT BIT RESOLUTION*

Assuming that the smart-choice algorithm was applied to all the *collapsed overlap regions* of the P - $\hat{\mathcal{S}}$ diagram, then there is a good chance that we might end up with a set of constants that has a maximum fractional-bit count less than n_P , which is the interesting phenomenon that we wish to discuss here.

This number, known in the code as `smart`constant`bits`, is computed as follows:

```
smart`constant`bits := -Log[2, GCD@@Join@@smart`constants];
```

In brief: the smart constant matrix is converted to a one-dimensional list through the application of the `Join` function, then, the *Greatest Common Divisor* is found using the `GCD` function, which returns the greatest *ulp* shared by all the constants. After that, the binary logarithm is found and negated (which is equivalent to finding the logarithm of the reciprocal).

For example, assuming we had the following set of comparison constants:

$$\left\{ \frac{11}{4}, \frac{13}{4}, 4, \frac{9}{2}, \frac{3}{2}, 2, 2, 3, \frac{1}{2}, 1, 1, 1 \right\}$$

then the previous expression would evaluate to 2, which means that two fractional bits are sufficient to represent any of these constants.

(note that n_P was equal to 4 in this case)

This implies that only 2 fractional bits would be needed to compare \hat{P} against each one of the constants, resulting in the correct choice of s'_i .

Yet, in an SRT-table-based system, no actual comparisons take place, rather, the table is indexed using the truncated pair $\{\hat{P}, \hat{\mathcal{S}}\}$, which returns the final outcome of the comparisons, stored as a static value in the table. Hence, the above translates to

reducing the number of fractional bits used in indexing the table from the original n_P to this reduced number, denoted as $n_{P_{smart}}$.

In the example above, a 75% reduction in the size of the SRT table can be achieved by reducing the number of fractional input bits from 4 to 2, which is significant saving.

Note that despite the reduction in the SRT-table size possible by smartly choosing the constants, the *ulp* of the truncated remainder \hat{P} is still assumed to be equal to 2^{-n_P} , which is used in the error calculations.

This means that in the case of a two-word representation of the remainder P (e.g. a *cary-save* or a *signed-bit* representation), the two words are still assumed to be both truncated to n_P fractional bits (rather than $n_{P_{smart}}$ bits), then added/subtracted in this truncated accuracy, using an appropriately-sized adder/subtractor.

3.2.5 Evolving precision of $\dot{\mathcal{S}}$ (Advanced)

Unlike the divisor D in the case of division (which is the amount used to index the table beside the remainder), the fractional result $\dot{\mathcal{S}}_{i-1}$ has an interesting property which is that it's continuously evolving in accuracy.

Namely, as one more digit gets appended to the result in every iteration, the number of different values associated with it increases.

In the fused algorithm, the current result \mathcal{S}'_{i-1} would have already accumulated $i - 1$ digits or $m(i - 1)$ bits, Z of which are zeros. This implies $m(i - 1) - Z$ actual result bits in the i 'th iteration, which is a positive number for iterations starting from, and following the $(\delta + 1)$ 'th iteration⁽¹⁰⁾.

Speaking of the fractional result $\dot{\mathcal{S}}_{i-1}$, having this variable number of fractional bits implies a decreasing *ulp*:

$$ulp(\dot{\mathcal{S}}_{i-1}) = \begin{cases} 0 & \text{for } i \leq \delta \\ 2^{-m(i-1)+Z} & \text{for } i > \delta \end{cases}$$

This has a number of consequences. Specifically, in iterations where the number of

⁽¹⁰⁾ Note that even though the first actual result digit is produced in the δ 'th iteration, this digit is not a part of the current result \mathcal{S}'_{i-1} until the $(\delta - 1)$ 'th iteration.

result bits is still less than or equal to n_S , the SRT table will be addressed using the exact result value. This means that the input \hat{S} in these iterations will be associated with only one, rather than a range of possible values.

Furthermore, in iterations where the number of fractional result bits is strictly less than n_S , the number of values that can be possessed by \hat{S} is limited, due to \dot{S}_{i-1} having a *ulp* greater than 2^{-n_S} .

Both observations can help optimize the overlap intervals injected by these, otherwise-problematic *early iterations*, which is one of the main techniques deployed in [8] (and followed here) to reduce the need for an initial PLA⁽¹¹⁾.

To ease referring to these iterations, we will define σ in a way similar to δ , as the first iteration in which more than n_S result bits are produced (Fig. 3.9).

It follows from the definition of σ that it would also be the last iteration in which the current result's fractional bits will be less than or equal to n_S .

Hence, the situation in the early iterations can be summarized as follows:

$$\boxed{\begin{array}{l} \delta < i \leq \sigma : \\ \dot{S} \in [\hat{S}, \hat{S}] \\ \hat{S} = k \times 2^{-m(i-1)+Z} \text{ where } k \in \text{Integers} \end{array}}$$

As for the iterations following the σ 's iteration, which we call the *late iterations*, knowledge of the changing *ulp* can still help refine the truncation intervals associated

⁽¹¹⁾ Remember that we're still advocating the use of some first-digit selection mechanism, in the final architecture in Chapter 4.

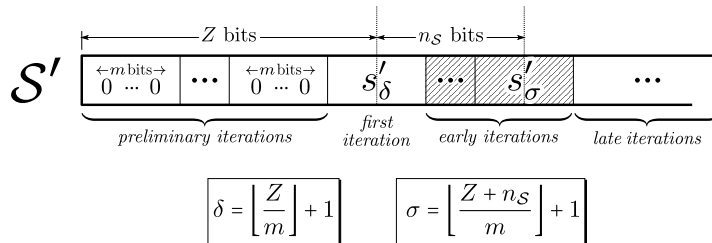


Figure 3.9: Illustration of preliminary, early, and late iterations, using the δ and the σ parameters. Note that the early iterations are highlighted.

with different values of $\hat{\mathcal{S}}$, as follows:

$$i > \sigma : \quad \begin{aligned} \dot{\mathcal{S}} &\in \left[\hat{\mathcal{S}}, \hat{\mathcal{S}} + 2^{-n_S} - 2^{-m(i-1)+Z} \right] \\ \hat{\mathcal{S}} &= k \times 2^{-n_S} \end{aligned}$$

Excluding impossible digit choices

An interesting implication of passing an exact result value in the early iterations is the fact that some digit choices, at the boundaries of the fractional result's standard range, will be deemed impossible.

For example, if $\dot{\mathcal{S}}$ had a standard range of $[\frac{1}{2}, 1)$, which is usually the case⁽¹²⁾, then negative digit values cannot be returned when $\dot{\mathcal{S}}$ is exactly $\frac{1}{2}$, as this would push the result outside its range, which is known to be impossible. In the same way, positive digit values (except zero) cannot be returned when $\dot{\mathcal{S}}$ is exactly 1, for the same reason.

Note that as $\hat{\mathcal{S}}$ turns into an approximation (in iterations following σ), this reasoning stops to hold since a negative digit value might be needed to correct a result in the range $[\frac{1}{2}, \frac{1}{2} + 2^{-n_S})$, by bringing it closer to $\frac{1}{2}$, but without necessarily pushing it outside its designated range, and vice versa.

Unfortunately, due to the result's extended range in the fused multiplication/square-root case, this technique is inapplicable at the lower end of the indexing range $[\frac{1}{2}, 1)$. Yet, it is still possible to exclude digit choices at the $(\delta+1)$ 'th iteration through a special design of the First-Digit Selector (this is the subject of subsection 4.2.6 on page 98).

⁽¹²⁾ More on this in section 4.2.

INTERVAL-INCLUSION ARRAY*

For every digit choice s'_i and a truncated result \hat{S} combination, the *Mathematica*[®] platform computes a number of overlap intervals:

1. $(\sigma - \delta)$ intervals for each one of the early iterations,
2. A single interval for all the following (i.e. *late*) iterations⁽¹³⁾.

These intervals are computed prior to the interactive operation, and hence cannot be altered dynamically after executing the script.

However, the code associates a binary flag with each one of these intervals, which conveys the user's wish either to include, or to exclude that particular interval from the final comparison-constant computations.

Namely, including or excluding an interval is as easy as checking or unchecking the checkbox next to it in the interactive, *designer cell* interface (Fig. 3.10).

Once you check or uncheck an interval, the final interval affected by it (the one in curly braces underneath) is updated instantly, with an exclamation mark \triangle next to it to indicate that the minimum-height requirements were not met by the interval, or an asterisk to indicate that the final interval was effectively nonexistent (page 54).

Note that this feature, which we call the *interval inclusion feature*, is used to automatically exclude the impossible digit choices, using the following logical test which is used to assign `False` to selected elements of the inclusion array `intervals`inclusion`:

```
intervals`inclusion[[Join @@ Position[
    labels, {digit_, index_, iteration_} /;
    (iteration <=  $\sigma$  & index == 1 &
    digit < 0)]]] = False;
];
```

⁽¹³⁾This interval is obtained by intersecting the intervals associated with all the following iterations, using the interval-intersecting technique on page 59.

	0 ($\hat{S}=\frac{1}{2}$)	1 ($\hat{S}=\frac{9}{16}$)	2 ($\hat{S}=\frac{5}{8}$)	3 ($\hat{S}=\frac{11}{16}$)
$i \geq 5$	✓ 40.0, 47.0	✓ 48.0, 59.0	✓ 56.0, 71.0	✓ 64.0, 83.0
$i = 4$	✓ 36.0, 51.5	✓ 52.0, 75.5		
$c_3[i]$	$c_3[1] = \left\{ \frac{21}{8}, \frac{45}{16} \right\}$	$c_3[2] = \left\{ \frac{53}{16}, \frac{57}{16} \right\}$	$c_3[3] = \left\{ \frac{29}{8}, \frac{69}{16} \right\}$	$c_3[4] = \left\{ \frac{33}{8}, \frac{81}{16} \right\}$
$i \geq 5$	✓ 20.0, 31.0	✓ 24.0, 39.0	✓ 28.0, 47.0	✓ 32.0, 55.0
$i = 4$	✓ 17.0, 33.0	✓ 25.0, 49.0		
$c_2[i]$	$c_2[1] = \left\{ \frac{11}{8}, \frac{15}{8} \right\}$	$c_2[2] = \left\{ \frac{13}{8}, \frac{19}{8} \right\}$	$c_2[3] = \left\{ \frac{15}{8}, \frac{23}{8} \right\}$	$c_2[4] = \left\{ \frac{17}{8}, \frac{27}{8} \right\}$
$i \geq 5$	✓ 0.0, 15.0	✓ 0.0, 19.0	✓ 0.0, 23.0	✓ 0.0, 27.0
$i = 4$	✓ 0.0, 15.5	✓ 0.0, 23.5		
$c_1[i]$	$c_1[1] = \left\{ \frac{1}{8}, \frac{7}{8} \right\}$	$c_1[2] = \left\{ \frac{1}{8}, \frac{9}{8} \right\}$	$c_1[3] = \left\{ \frac{1}{8}, \frac{11}{8} \right\}$	$c_1[4] = \left\{ \frac{1}{8}, \frac{13}{8} \right\}$
$i \geq 5$	✓ 1.0, 15.8	✓ 1.0, 19.8	✓ 1.0, 23.8	✓ 1.0, 27.8
$i = 4$	✓ 1.0, 15.0	✓ 1.0, 23.0		
$c_0[i]$	$c_0[1] = -$	$c_0[2] = -$	$c_0[3] = -$	$c_0[4] = -$
$i \geq 5$	✓ 21.0, 31.0	✓ 25.0, 39.0	✓ 29.0, 47.0	✓ 33.0, 55.0
$i = 4$	16.5, 28.0	✓ 24.5, 44.0		
$c_{-1}[i]$	$c_{-1}[1] = -$	$c_{-1}[2] = -$	$c_{-1}[3] = -$	$c_{-1}[4] = -$
$i \geq 5$	✓ 41.0, 45.8	✓ 49.0, 57.8	✓ 57.0, 69.8	✓ 65.0, 81.8
$i = 4$	31.0, 39.0	✓ 47.0, 63.0		
$c_{-2}[i]$	$c_{-2}[1] = -$	$c_{-2}[2] = -$	$c_{-2}[3] = -$	$c_{-2}[4] = -$

Figure 3.10: The interactive, *designer cell* interface of the *Mathematica*[®] platform.

3.2.6 Unsigned SRT table (Advanced)

Unlike the case of division, where the selection regions in the negative half exhibit a clear symmetry with those in the positive half, this so-called *symmetry* is disturbed in the case of the square root by the terms $2^Z r^{-i} (s'_i \pm h_B^\pm)^2$ in (3.6).

Yet, it's still possible to combine the overlap regions in the negative half of the $P\text{-}\dot{S}$ diagram with their equivalents in the positive half, through the mechanism of intersecting the intervals (page 59).

This, if led to feasible final intervals, will allow us to cut the size of the SRT table in half, by storing only the digit-selection information corresponding to the positive half of the $P\text{-}\dot{S}$ diagram.

As for negative values of the remainder, which no longer have digit-selection information associated with them, they are first mapped to the positive half of the $P\text{-}\dot{S}$ plane by the act of negation, then used to address the table. Note that in this case, the resulting digit has also to be negated.

For this scheme to work, it is required that the upper and lower lines of the negative overlap regions be negated and swapped, since the lower line will become an upper line, and the upper line will become a lower line (Fig. 3.11).

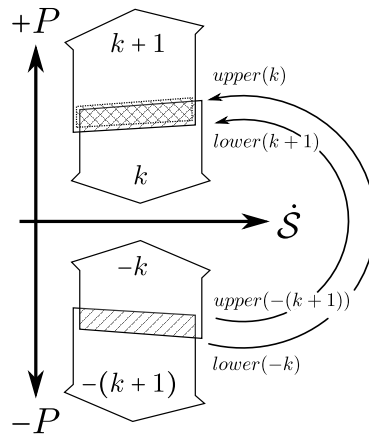


Figure 3.11: Folding the overlap regions into the positive half of the $P\text{-}\dot{S}$ diagram resulting in combined, *unsigned overlap regions*.

Furthermore, as shown in the figure, the overlap region delimiting the $(-k)$ 'th digit choice from beneath will, upon negation (i.e. *folding* into the positive half) delimit the

k 'th digit choice from above, which means that it would (in its *folded form*) correspond to the $(k + 1)$ 'th digit.

The only problem with this technique is the part of negating P before using it to address the SRT table. This negation corresponds to finding the 2's complement of the number (prior to truncating it), which is an expensive, carry-propagating operation.

Luckily, in a practical system where the residual is maintained in a carry-save form, a limited-precision addition is already needed to produce the truncated remainder input \hat{P} of the table. Hence, this act of addition can be used to produce a negated copy of \hat{P} , without increasing the delay of the circuit. This comes at the expense of duplicating the limited-precision adder as shown in Fig. 3.12.

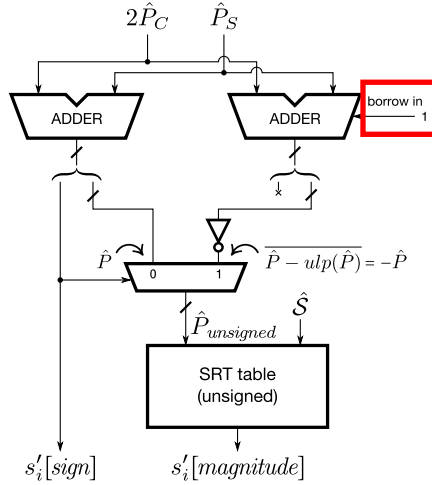


Figure 3.12: Hardware needed for an unsigned SRT table.

As shown in the figure, deploying an unsigned SRT table incurs some extra hardware at the input of the table, which adds to the delay of the digit-selection path due to inverting and multiplexing the table input. Furthermore, using an unsigned SRT table dictates that s'_i will be produced in a sign-and-magnitude format.

3.2.7 Custom mappings (Advanced)

In subsection 3.2.5, we discussed ways to mitigate the negative impact of the early iterations on the overlap regions, through a combination of techniques such as: refined truncation intervals for the \hat{S} input, a reduced target set for this input in the early

iterations, and the exclusion of impossible digit choices at the boundaries of the result's standard range.

Yet, even with all those techniques, it's still sometimes the case that the intervals injected by one of the early iterations would seriously restrict the final overlap regions for one of the $\hat{\mathcal{S}}$ values.

Luckily, this situation can be fixed by manipulating the value of $\hat{\mathcal{S}}$ before using it to address the table, which can be seen as mapping the problematic intervals to another $\hat{\mathcal{S}}$ range where they would be less harmful.

Note that even though such mappings can reduce the internal complexity of the SRT table, they do require additional combinational logic at the input, which will add to the delay of the digit-selection path.

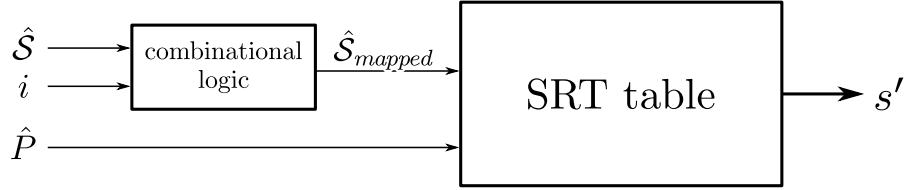


Figure 3.13: The *mapping logic* at the input of the SRT table.

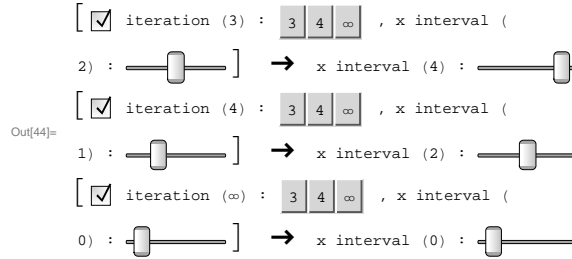


Figure 3.14: A snapshot showing the custom mapping interface in the software platform.

3.3 Sharing the SRT Table with Division

As shown in section A.2 in the appendix, digit-selection in fused multiplication-division is bound to the following correctness constraint, which accepts the selection remainder P (an amount equivalent to the selection remainder in the case of the square

root) and the fractional divisor \dot{D} as its inputs (A.8):

$$\dot{D}(q'_i - h^-) + 2^{-Z}h_B^- < P_{i-1} < \dot{D}(q'_i + h^+) - 2^{-Z}h_B^+ \quad (3.7)$$

By comparing this to the final correctness constraint in (3.6):

$$\begin{aligned} 2\dot{\mathcal{S}}_{i-1}(s'_i - h^-) + 2^{-Z}h_B^- &< P_{i-1} < 2\dot{\mathcal{S}}_{i-1}(s'_i + h^+) - 2^{-Z}h_B^+ \\ + 2^Z r^{-i}(s'_i - h^-)^2 && + 2^Z r^{-i}(s'_i + h^+)^2 \end{aligned}$$

we can see that if we drop the iteration-dependent term, then substitute \dot{D} for $2\dot{\mathcal{S}}_{i-1}$, we can actually translate our correctness constraint to that of fused multiplication/division, while noting that the next square-root digit s'_i will become the next quotient digit q'_i .

As for dropping the iteration-dependent term, this is equivalent to supporting an infinite number of iterations, by evaluating the overlap regions at an infinite (∞) iteration number (which is supported in the *Mathematica*[®] script), then intersecting these regions with the overlap regions of all the other “actual” algorithm iterations.

This will result in the following, infinite-iteration correctness constraint:

$$2\dot{\mathcal{S}}_{i-1}(s'_i - h^-) + 2^{-Z}h_B^- < P_{i-1} < 2\dot{\mathcal{S}}_{i-1}(s'_i + h^+) - 2^{-Z}h_B^+ \quad (3.8)$$

which is quite similar to (3.7), with the only difference being multiplying the input $\dot{\mathcal{S}}/\dot{D}$ by two in the first term.

Namely, this correctness constraint models the operation $\frac{A \times B}{2D}$ rather than $\frac{A \times B}{D}$, which is the intended fused operation in the case of division.

The solution to this is simple, which is a combination of the following:

1. Passing $\frac{1}{2}\dot{D}$ to the SRT table.
2. Using \dot{D} in the fused recurrence rather than $\frac{1}{2}\dot{D}$ to update the residual.
3. Adjusting the normalization of \dot{D} to ensure that $\frac{1}{2}\dot{D}$ has the same domain as $\dot{\mathcal{S}}$, which is typically $[\frac{1}{2}, 1)$.

Hence, some modifications are needed to allow the fused multiplication/division unit to share the SRT table with a fused multiplication/square-root unit, but they are minor modifications.

Note that the fused multiplication/division unit has obviously to be designed around the same set of system parameters, such as the radix, the same internal digit set, the same executional delay Z , etc.

3.3.1 Effect on $n_{Pintegral}$

In this thesis, the SRT table designed for fused multiplication/square-root is shared with fused multiplication/division through passing half of the divisor $\frac{1}{2}\dot{D}$ to the table. This is unlike other studies [16] in which the SRT table of division is shared with a square-root unit by passing two times the result $2\dot{S}_{i-1}$.

One of the key consequences of this difference is the impact on $n_{Pintegral}$, which is a function of \hat{P}_{max} and \hat{P}_{min} .

Namely, due to the multiplication by 2 in the upper and lower bounds of (3.8) as compared to (3.7), the maximum and the minimum values of the remainder input \hat{P} will be doubled, dictating an increased integral bit count of one.

This should be taken into consideration when comparing the truncation pairs $\{n_P, n_S\}$ of fused multiplication/square-root with those associated with a fused multiplication/-division design, by incrementing the effective n_P value by one for a fair comparison.

3.4 Generating the SRT table contents

So far, we have discussed the process of obtaining the comparison constants, but neglected what to do with these constants, or how to arrive at the actual device that will be used for digit selection.

Unlike the arithmetic model, in which the value of \hat{P} is actually compared against each one of the $\alpha + \beta$ constants for the relevant \hat{S} value, to arrive at a valid digit choice, the overwhelming majority of designs today are based on obtaining the next digit choice

in constant time, through the use of a look-up table.

This means that a table will be created with entries for all the different \hat{P}/\hat{S} combinations, given the *ulp* of either amount⁽¹⁴⁾, then, based on the number of the \hat{S} constants equalled or exceeded by the value of \hat{P} , we will store a “static” value in that cell that corresponds to the implied digit choice.

This way, the delay of enquiring the table will be independent from the criteria underlying the production of its contents, which is the main idea.

3.4.1 A design example

Our typical example of a practical fused system in this thesis is a maximally-redundant, radix-4 system with a Z value of $4^{(15)}$, and an *actual first selection iteration* of 4 (rather than 3, which is the δ 'th iteration implied by these parameters).

The ability to skip the table in the δ 'th iteration is achieved through another device called the First-Digit Selector, whose design and purpose will be explored in the next chapter. For the time being, we will just accept the fact that the table can be designed while excluding some of the iterations that require some means of digit selection.

This system assumes an n_P value of 4 fractional bits, an n_S value of 4 fractional bits, an irredundant multiplier B , an unsigned SRT table, and no mappings (Fig. 3.15).

These parameters result in the overlap intervals in Fig. 3.16, and the set of comparison constants in the table below (remember that these constants are for an unsigned table):

	$\hat{S} = 0.100_2$	$\hat{S} = 0.101_2$	$\hat{S} = 0.110_2$	$\hat{S} = 0.111_2$
$ s'_i = 3$	$\frac{11}{4}$	$\frac{7}{2}$	4	5
$ s'_i = 2$	$\frac{3}{2}$	2	2	3
$ s'_i = 1$	$\frac{1}{2}$	1	1	1

Thanks to the smart choice of the comparison constants, the constants above are all multiples of $\frac{1}{4}$, implying an $n_{P_{smart}}$ value of 2 bits only. This results in the compact SRT table contents in Table 3.1.

⁽¹⁴⁾ Note that in case we were able to obtain an $n_{P_{smart}}$ value less than n_P , then we can design the table around this increased *ulp* of $2^{-n_{P_{smart}}}$, rather than the actual *ulp* which is 2^{-n_P} .

⁽¹⁵⁾ A theoretical lower bound on the value of Z will be derived in subsection 4.2.5 on page 97.

Table 3.1: The SRT table contents of our maximally-redundant, radix-4 system. Note that this SRT table is unsigned, and can also be shared with a used multiplication/division unit (i.e. has support for a theoretical, infinite iteration).

	$\hat{\mathcal{S}} = \frac{1}{2}$	$\hat{\mathcal{S}} = \frac{5}{8}$	$\hat{\mathcal{S}} = \frac{3}{4}$	$\hat{\mathcal{S}} = \frac{7}{8}$
$\hat{P}_{unsigned} = 8$	*	*	*	3
$\hat{P}_{unsigned} = \frac{31}{4}$	*	*	*	3
$\hat{P}_{unsigned} = \frac{15}{2}$	*	*	*	3
$\hat{P}_{unsigned} = \frac{29}{4}$	*	*	*	3
$\hat{P}_{unsigned} = 7$	*	*	3	3
$\hat{P}_{unsigned} = \frac{27}{4}$	*	*	3	3
$\hat{P}_{unsigned} = \frac{13}{2}$	*	*	3	3
$\hat{P}_{unsigned} = \frac{25}{4}$	*	*	3	3
$\hat{P}_{unsigned} = 6$	*	3	3	3
$\hat{P}_{unsigned} = \frac{23}{4}$	*	3	3	3
$\hat{P}_{unsigned} = \frac{11}{2}$	*	3	3	3
$\hat{P}_{unsigned} = \frac{21}{4}$	*	3	3	3
$\hat{P}_{unsigned} = 5$	3	3	3	3
$\hat{P}_{unsigned} = \frac{19}{4}$	3	3	3	2
$\hat{P}_{unsigned} = \frac{9}{2}$	3	3	3	2
$\hat{P}_{unsigned} = \frac{17}{4}$	3	3	3	2
$\hat{P}_{unsigned} = 4$	3	3	3	2
$\hat{P}_{unsigned} = \frac{15}{4}$	3	3	2	2
$\hat{P}_{unsigned} = \frac{7}{2}$	3	3	2	2
$\hat{P}_{unsigned} = \frac{13}{4}$	3	2	2	2
$\hat{P}_{unsigned} = 3$	3	2	2	2
$\hat{P}_{unsigned} = \frac{11}{4}$	3	2	2	1
$\hat{P}_{unsigned} = \frac{5}{2}$	2	2	2	1
$\hat{P}_{unsigned} = \frac{9}{4}$	2	2	2	1
$\hat{P}_{unsigned} = 2$	2	2	2	1
$\hat{P}_{unsigned} = \frac{7}{4}$	2	1	1	1
$\hat{P}_{unsigned} = \frac{3}{2}$	2	1	1	1
$\hat{P}_{unsigned} = \frac{5}{4}$	1	1	1	1
$\hat{P}_{unsigned} = 1$	1	1	1	1
$\hat{P}_{unsigned} = \frac{3}{4}$	1	0	0	0
$\hat{P}_{unsigned} = \frac{1}{2}$	1	0	0	0
$\hat{P}_{unsigned} = \frac{1}{4}$	0	0	0	0
$\hat{P}_{unsigned} = 0$	0	0	0	0

Bits per high-radix digit (2) :

α (3) :

β (3) :

B^+ (3) :

B^- (0) :

n^x (3) :

n^p (4) :

Z (4) :

Out[183]= Single Precision Double Precision Quadruple Precision

☒ Share SRT table with fused multiplication/division (Yes)

☒ Support a carry-save residual (Yes)

☒ Design an unsigned SRT table (Yes)

First selection iteration (4) :

Number of interval mappings to be used (0) :

Figure 3.15: The system-parameter view of our maximally-redundant, radix-4 system.

	0 ($\hat{S}=\frac{1}{2}$)	1 ($\hat{S}=\frac{9}{16}$)	2 ($\hat{S}=\frac{5}{8}$)	3 ($\hat{S}=\frac{11}{16}$)
$i \geq 5$	<input checked="" type="checkbox"/> 40.0,47.0	<input checked="" type="checkbox"/> 48.0,59.0	<input checked="" type="checkbox"/> 56.0,71.0	<input checked="" type="checkbox"/> 64.0,83.0
$i = 4$	<input checked="" type="checkbox"/> 36.0,51.5	<input checked="" type="checkbox"/> 52.0,75.5		
$c_3[i]$	$c_3[1]=\left\{\frac{21}{8}, \frac{45}{16}\right\}$	$c_3[2]=\left\{\frac{53}{16}, \frac{57}{16}\right\}$	$c_3[3]=\left\{\frac{29}{8}, \frac{69}{16}\right\}$	$c_3[4]=\left\{\frac{33}{8}, \frac{81}{16}\right\}$
$i \geq 5$	<input checked="" type="checkbox"/> 20.0,31.0	<input checked="" type="checkbox"/> 24.0,39.0	<input checked="" type="checkbox"/> 28.0,47.0	<input checked="" type="checkbox"/> 32.0,55.0
$i = 4$	<input checked="" type="checkbox"/> 17.0,33.0	<input checked="" type="checkbox"/> 25.0,49.0		
$c_2[i]$	$c_2[1]=\left\{\frac{11}{8}, \frac{27}{16}\right\}$	$c_2[2]=\left\{\frac{13}{8}, \frac{19}{8}\right\}$	$c_2[3]=\left\{\frac{15}{8}, \frac{23}{8}\right\}$	$c_2[4]=\left\{\frac{17}{8}, \frac{27}{8}\right\}$
$i \geq 5$	<input checked="" type="checkbox"/> 0.0,15.0	<input checked="" type="checkbox"/> 0.0,19.0	<input checked="" type="checkbox"/> 0.0,23.0	<input checked="" type="checkbox"/> 0.0,27.0
$i = 4$	<input checked="" type="checkbox"/> 0.0,15.5	<input checked="" type="checkbox"/> 0.0,23.5		
$c_1[i]$	$c_1[1]=\left\{\frac{1}{8}, \frac{7}{8}\right\}$	$c_1[2]=\left\{\frac{1}{8}, \frac{9}{8}\right\}$	$c_1[3]=\left\{\frac{1}{8}, \frac{11}{8}\right\}$	$c_1[4]=\left\{\frac{1}{8}, \frac{13}{8}\right\}$
$i \geq 5$	<input checked="" type="checkbox"/> 1.0,15.8	<input checked="" type="checkbox"/> 1.0,19.8	<input checked="" type="checkbox"/> 1.0,23.8	<input checked="" type="checkbox"/> 1.0,27.8
$i = 4$	<input checked="" type="checkbox"/> 1.0,15.0	<input checked="" type="checkbox"/> 1.0,23.0		
$c_0[i]$	$c_0[1]=-$	$c_0[2]=-$	$c_0[3]=-$	$c_0[4]=-$
$i \geq 5$	<input checked="" type="checkbox"/> 21.0,31.0	<input checked="" type="checkbox"/> 25.0,39.0	<input checked="" type="checkbox"/> 29.0,47.0	<input checked="" type="checkbox"/> 33.0,55.0
$i = 4$	<input checked="" type="checkbox"/> 16.5,28.0	<input checked="" type="checkbox"/> 24.5,44.0		
$c_{-1}[i]$	$c_{-1}[1]=-$	$c_{-1}[2]=-$	$c_{-1}[3]=-$	$c_{-1}[4]=-$
$i \geq 5$	<input checked="" type="checkbox"/> 41.0,45.8	<input checked="" type="checkbox"/> 49.0,57.8	<input checked="" type="checkbox"/> 57.0,69.8	<input checked="" type="checkbox"/> 65.0,81.8
$i = 4$	<input type="checkbox"/> 31.0,39.0	<input checked="" type="checkbox"/> 47.0,63.0		
$c_{-2}[i]$	$c_{-2}[1]=-$	$c_{-2}[2]=-$	$c_{-2}[3]=-$	$c_{-2}[4]=-$

Figure 3.16: The overlap intervals associated with our radix-4 system.

3.5 The Software Platform

So far, we have used our custom, *Mathematica*[®]-based tools to design the contents of the SRT table, without elaborating much on the organization of the tools, and how to use them to explore the design space of the table.

The tools developed as part of this research are:

1. **Selection function workbench** for illustrating the selection ranges (not used for the actual design of the table). The purpose of this tool is to help the designer understand the effect of changing the different parameters on the overlap regions, and identify the most helpful changes.
2. **Selection function designer** for designing the SRT table. This tool is composed of three modules:
 - (a) The **systems parameters** module.
 - (b) The **designer cell** module.
 - (c) The **look-up table generator** module.
3. **Exact and SRT-table solvers** for demonstrating the solution process with or without an SRT table (this primitive simulation tool is replaced in this thesis by the C-language-based, bit-level simulator, which will be presented in Chapter 5).

Note that the first and the third scripts above have played a historical role in helping the author gain a better understanding of the SRT algorithms. That is to say, beside their historical role, they are not important for the results of this thesis.

The second script, on the other hand, which is composed of three different modules, is of central importance as it comprises the main tool used in this thesis to arrive at a set of good system parameters, and design the SRT table according to these parameters.

To explore the design space using this tool, we will follow an exploration cycle similar to the one in Fig. 3.17. The final output of this cycle in our case is a C-code fragment that fully characterizes the system, which can be used to conduct bit-level simulations of the algorithm (more on this in Chapter 5).

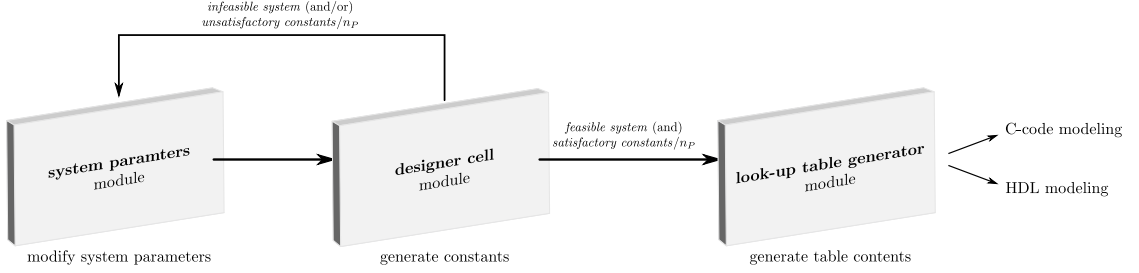


Figure 3.17: The design cycle dictated by the organization of the SRT-table-design platform.

3.5.1 Exploration Strategy

To find good designs, we start by choosing the radix, represented by the number of result bits retired in every iteration m . Hence, the parameter m lies at the top of the parameter hierarchy associated with good design-space exploration.

After determining m , we decide the digit set of the result, represented by the choice of the parameters h^+ and h^- .

To test the basic feasibility of the chosen $m/h^+/h^-$ configuration, we compute the overlap intervals using an irredundant multiplier B ($B^+ = r - 1, B^- = 0$), an initial value for n_S of 5 or 4, an arbitrarily-high value for n_P and Z , and no features (i.e. no support for a carry-save residual, an unsigned table, no mappings, etc.).

Note that preferably, we will begin our search with the assumption that a Second-Digit Selector is employed, by specifying a first selection iteration of $\delta + 3$.

The logic of this step is to give the desired $m/h^+/h^-$ configuration the maximum chance of resulting in a feasible set of comparison constants, while starting to “tighten” each one of the parameters in a wise order afterwards, to reach an optimal set of system parameters.

If the overlap intervals associated with this *entry-point configuration* didn’t lead to a feasible set of parameters, we analyze whether excluding another iteration would solve the problem, or whether the problematic intervals can be mapped to another \hat{S} interval where they would be less harmful.

Once we reach a working entry-level configuration, which results in identifying the right number of higher-order digit selectors or mappings, we see if we can reduce the

value of n_S while still leading to feasible constants.

After that, we activate the support for a carry-save adder and the sharing of the SRT table, and observe the effect on the intervals.

If all was good, we find the minimum value of Z that doesn't disturb the feasibility of the intervals.

Then, we reduce the value of n_P to the computed value of $n_{P_{smart}}$, which can either lead to a feasible set or not, if not, we increment n_P till a feasible set is reached⁽¹⁶⁾.

Note that as we experiment with the value of n_P , and observe the computed value of $n_{P_{smart}}$, we also try the option of an unsigned SRT table for each one of these values, to further explore this size-reducing possibility (Fig. 3.18)

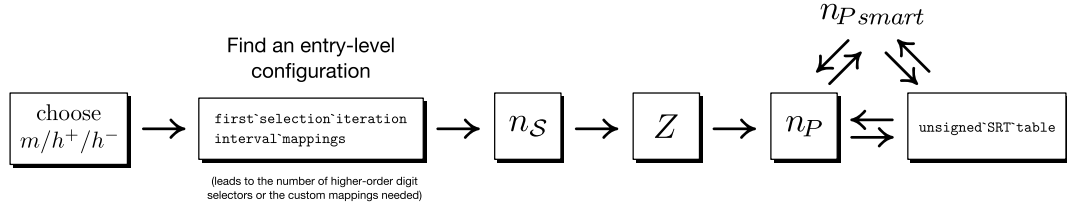


Figure 3.18: An illustration of the suggested strategy for the SRT table design-space exploration. Note that each box corresponds to determining one of the system parameter, whereas arrows dictate the order of such determination of parameters.

3.5.2 Good Designs

By following the strategy of the previous subsection, we reach the set of good designs summarized in Table 3.2.

In all of those designs, the feasibility of the SRT table was not affected by the choice of the multiplier's digit set. Namely, it is possible either to go for a minimally-redundant or an irredundant multiplier B in all of those cases, where the latter choice corresponds to generating fewer multiples of A .

Note that in cases where $n_{P_{smart}}$ was equal to n_P , only the former value was given, whereas the total number of input bits was noted in the leftmost column of the table, to

⁽¹⁶⁾ Note that sometimes, a value higher than the minimum n_P value can result in a reduced computed value of $n_{P_{smart}}$, which corresponds to a smaller total number of input bits and consequently, a smaller table (at the expense of a larger adder for computing the remainder sample \hat{P}).

ease tracking the size of the resulting table.

Table 3.2: Some of the good designs that can be obtained using our SRT-table-design platform.

Total input bits					Higher-Order Digit Selectors	Carry-Save	Shared table	Unsigned table	Custom Mappings
	$m = 2$ (radix 4)								
10	$n_S = 4$	$n_P = 3$ $n_{P_{integral}} = 3$	$h^\pm = \frac{2}{3}$ $h_B^\pm = *$	$Z = 4$	Second-Digit Selector	No	No	No	No need
10	$n_S = 4$	$n_P = 4$ $n_{P_{integral}} = 3$	$h^\pm = \frac{2}{3}$ $h_B^\pm = *$	$Z = 6$	No need	Yes	Yes	Yes	$(i = 5)$ $\hat{S}:1 \rightarrow \hat{S}:4$
9	$n_S = 4$	$n_P = 6, n_{P_{smart}} = 3$ $n_{P_{integral}} = 3$	$h^\pm = \frac{2}{3}$ $h_B^\pm = *$	$Z = 6$	No need	Yes	Yes	Yes	$(i = 5)$ $\hat{S}:1 \rightarrow \hat{S}:4$
9	$n_S = 3$	$n_P = 3$ $n_{P_{integral}} = 4$	$h^\pm = 1$ $h_B^\pm = *$	$Z = 4$	No need	Yes	Yes	Yes	No need
8	$n_S = 3$	$n_P = 4, n_{P_{smart}} = 2$ $n_{P_{integral}} = 4$	$h^\pm = 1$ $h_B^\pm = *$	$Z = 4$	No need	Yes	Yes	Yes	No need
	$m = 3$ (radix 8)								
16	$n_S = 6$	$n_P = 7, n_{P_{smart}} = 6$ $n_{P_{integral}} = 4$	$h^\pm = \frac{4}{7}$ $h_B^\pm = *$	$Z = 9$	Second-Digit Selector	Yes	Yes	No	No need
11	$n_S = 5$	$n_P = 5, n_{P_{smart}} = 3$ $n_{P_{integral}} = 4$	$h^\pm = \frac{5}{7}$ $h_B^\pm = *$	$Z = 6$	Second-Digit Selector	Yes	Yes	Yes	No need
12	$n_S = 6$	$n_P = 3$ $n_{P_{integral}} = 4$	$h^\pm = \frac{5}{7}$ $h_B^\pm = *$	$Z = 6$	Second-Digit Selector	Yes	Yes	Yes	No need
10	$n_S = 5$	$n_P = 2$ $n_{P_{integral}} = 4$	$h^\pm = 1$ $h_B^\pm = *$	$Z = 6$	Second-Digit Selector	Yes	Yes	Yes	No need
	$m = 4$ (radix 16)								
14	$n_S = 5$	$n_P = 6, n_{P_{smart}} = 5$ $n_{P_{integral}} = 5$	$h^\pm = \frac{14}{15}$ $h_B^\pm = *$	$Z = 8$	Second-Digit Selector	Yes	Yes	Yes	No need
12	$n_S = 6$	$n_P = 2$ $n_{P_{integral}} = 5$	$h^\pm = 1$ $h_B^\pm = *$	$Z = 8$	Second-Digit Selector	Yes	Yes	Yes	No need
	$m = 5$ (radix 32)								
14	$n_S = 7$	$n_P = 2$ $n_{P_{integral}} = 6$	$h^\pm = 1$ $h_B^\pm = *$	$Z = 10$	Second-Digit Selector	Yes	Yes	Yes	No need

Chapter 4

Hardware Design

4.1 A Register-Based Algorithm Description

In this section, we will change the final description of the fused algorithm (2.30) on page 45, by substituting implementation-friendly amounts for some of the variables in it. This will facilitate computing the register sizes, deriving sampling bit positions, and alignment of adders/subtractors, all of which are essential tasks to the hardware design mission carried in this chapter.

To reduce notational noise, and to signify the direct correspondence of these new amounts with physical devices or wires in the implementation, we will denote them using a computerized font similar to the one used for the *Mathematica*[®] code in Chapter 3.

As a start, we will define three digital variables to correspond with each one of the three main registers maintained throughout the algorithm:

- \mathbf{S} : corresponds to the result register.
- \mathbf{A}^{\leftarrow} : corresponds to the shifted-multiplicand register.
- \mathbf{W} : corresponds to the residual register.

where the design convention will be to make both \mathbf{S} and \mathbf{A}^{\leftarrow} of the integral type, while making \mathbf{W} of the fixed-binary-point type.

This leads to the following definitions and substitutions:

$$\begin{aligned}
\mathbf{S}_j &= \overline{\mathcal{S}}'_j & \boxed{\overline{\mathcal{S}}'_j \leftarrow \mathbf{S}_j} \\
\mathbf{A}_j^\leftarrow &= \frac{A_j^\leftarrow}{r^{a-n}} & \boxed{A_j^\leftarrow \leftarrow \mathbf{A}_j^\leftarrow r^{a-n}} \\
\mathbf{W}_j &= \frac{W_j}{r^{2a}} & \boxed{W_j \leftarrow \mathbf{W}_j r^{2a}}
\end{aligned} \tag{4.1}$$

Note that these definitions are independent of the iteration number i , that is to say, they are merely intended to change the position of the binary point, in order to make tracking the lower end of numbers easier (in the case of \mathbf{S} and \mathbf{A}^\leftarrow), or facilitate the correct alignment between terms and words in the implementation (in the case of \mathbf{W}).

In addition to these substitutions, we will define the digital inputs \mathbf{A} and \mathbf{B} as the integers corresponding with A and B , as follows:

$$\begin{aligned}
\mathbf{A} &= \frac{A}{r^{a-n}} & \boxed{A \leftarrow \mathbf{A} r^{a-n}} \\
\mathbf{B} &= \frac{B}{r^{a-n}} & \boxed{B \leftarrow \mathbf{B} r^{a-n}}
\end{aligned} \tag{4.2}$$

By putting these substitutions into place in (2.30), we get:

$$\begin{aligned}
&\mathbf{S}_0 = 0 \\
&\mathbf{W}_0 = b_1 \mathbf{A} \times r^{-n-1} 2^{-2Z} \\
&\mathbf{A}_0^\leftarrow = \mathbf{A} \\
&1 \leq i \leq \text{iterations} \quad \left\{ \begin{aligned} &\mathbf{S}_i = r \mathbf{S}_{i-1} + s'_i \\ &\mathbf{W}_i = r^2 \mathbf{W}_{i-1} - s'_i \times \boxed{\mathbf{S}_{i-1} \mid 0 \mid s'_i} + b_{i+1} \mathbf{A}_{i-1}^\leftarrow \times r^{-n} 2^{-2Z} \\ &\mathbf{A}_i^\leftarrow = r \mathbf{A}_{i-1}^\leftarrow \end{aligned} \right. \tag{4.3} \\
&\mathcal{S}_{final} = \mathbf{S}_{iterations} \times r^{a-n} 2^{m_b} \\
&\mathcal{R}_{final} = \mathbf{W}_{iterations} \times r^{2a-2iterations} 2^{2Z}
\end{aligned}$$

Note that the total number of iterations was also changed from our previous formula $n^* + \delta - 1$ to **iterations**, since some implementation details will now play a role in

determining the number of needed iterations (this will be the subject of section 4.4).

Note that most of the discussions of this chapter will revolve around this implementation-friendly, register-based description of the algorithm.

4.1.1 Register charts

An important aspect of the resulting hardware algorithm is the number of bits needed for each one of the three registers, as well as the position of the binary point relative to the lower end of the register, which results in what we call the *register charts*.

The S register

The S register stores the value of the bare result $\overline{\mathcal{S}}'_j$, which is the integral equivalent of the delayed result \mathcal{S}'_j .

Namely, at the end of the last iteration, “`iterations` \times m ” bits would have been already pushed into this register (from its lower end), with the Z first bits being zero. This implies a bit-length requirement of “ $m(\text{iterations}) - Z$ ”, and an integral definition:

$$\text{S} \quad \overbrace{\hspace{10em}}^{m(\text{iterations}) - Z} \bullet$$

The A^{\leftarrow} register

As for the A^{\leftarrow} register, it stores the integral equivalent of the once-called the *shifted multiplicand* A_j^{\leftarrow} . Namely, it is nothing but the integral **A** which has been pushed to the left by one digit position per iteration for all the iterations from 1 to `iterations` - 1:

$$\begin{array}{c} A_0^{\leftarrow} \quad \overbrace{\hspace{10em}}^{\text{A}} \\ A_1^{\leftarrow} \quad \overbrace{\hspace{10em}}^{\text{A}} \\ A_2^{\leftarrow} \quad \overbrace{\hspace{10em}}^{\text{A}} \\ \dots \\ A_{\text{iterations}-1}^{\leftarrow} \quad \overbrace{\hspace{10em}}^{\text{A}} \\ A^{\leftarrow} \quad \overbrace{\hspace{10em}}^{m(\text{iterations} + n - 2)} \bullet \end{array}$$

$\hspace{10em} \underbrace{\hspace{10em}}_{mn} \hspace{10em} \underbrace{\hspace{10em}}_{m(\text{iterations} - 2)}$

Note that unlike the S register, the final value of the A register is not needed, and hence is dropped from the bit-length calculations.

The W register

Unlike the other two registers, the W register has an integral and a fractional parts, where it is easiest to obtain the bit-length of either part by taking a quick look at its initialization and update lines in (4.3):

$$W_0 = b_1 A \times r^{-n-1} 2^{-2Z}$$

$$W_i = r^2 W_{i-1} - s'_i \times \boxed{S_{i-1} \mid 0 \mid s'_i} + b_{i+1} A_{i-1}^{\leftarrow} \times r^{-n} 2^{-2Z}$$

Note that the update of W_i involves the subtraction of a linear-quadratic term and the addition of a partial-product term, whereas the initialization involves only the partial-product term.

Interestingly, the linear-quadratic term is a purely-integral amount:

$$s'_i \times \boxed{S_{i-1} \mid 0 \mid s'_i}$$

that is subtracted in every iteration to keep the value of the residual bounded.

Henceforth, the residual's integral part cannot be greater than the maximum value of this term, as this would mean that the error in the result is no longer under the control of the future result digits, which implies a *diverging* result.

This little fact can be used indirectly to obtain the integral-part bit length of W, by computing the maximum number of bits needed to represent this term.

This integral-bit count is $m(\text{iterations} + 1) - Z + 1$, which can be broken down as:

$$\begin{array}{c} \text{add } m \text{ bits} \quad m+1 \text{ bits} \\ s'_i \times \boxed{S_{i-1} \mid 0 \mid s'_i} \\ \underbrace{\hspace{1.5cm}}_{m(\text{iterations}-1)-Z \text{ bits}} \end{array} = m + 1 + m(\text{iterations} - 1) - Z + m$$

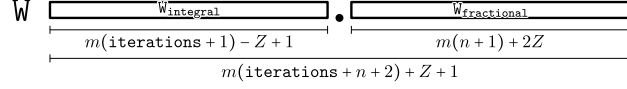
As for W's fractional-part bit length, it is determined by the initialization line above:

$$W_0 = b_1 A \times r^{-n-1} 2^{-2Z}$$

where the assigned value is nothing but $b_1 A$, an integer, whose ones' digit is shifted down

to the $m(n+1) + 2Z$ fractional-bit position (dictated by $r^{-n-1}2^{-2Z} = \frac{1}{2^{m(n+1)+2Z}}$).

This results in the following chart for the W register:



where the abbreviations $\text{W}_{\text{integral}}$ and $\text{W}_{\text{fractional}}$ can be conveniently used to refer to the integral and the fractional segments of the W register.

Note that even though the W register is somewhat wide, the adders/subtractors (or practically, the *compressors*) required to operate on it need not be as wide. A discussion of the processor widths is part of section 4.5.

4.2 Loose Leading Bit of $\dot{\mathcal{S}}$

So far, we have overlooked a critical aspect of the fused algorithm, represented by the result's leading bit position being uncertain.

Not only that the extended $\dot{\mathcal{S}}$ range associated with such uncertainty would double the size and the complexity of the SRT table, it would also cancel the advantage possibly gained by sharing the SRT table with division (since the table has to be doubled in size in order to be shared).

In fact, it would not be an exaggeration to say that this extended range, if not dealt with, would impair the *raisons d'être* of a fused multiplication/square-root unit.

4.2.1 Range of $\dot{\mathcal{S}}$ in a square-root unit

Before getting into the details of the problem, let's examine the case in pure square-root implementations. In this case, the radicand N is presented in a normalized form $\dot{N} \in [\frac{1}{2}, 1)$, while associated with a binary exponent N_{exp} .

To account for the case when N_{exp} is not an even number, a single binary shift of \dot{N} might be necessary, thus extending the range of this amount to $[\frac{1}{4}, 1)$, which corresponds to a square-root $\dot{\mathcal{S}}$ range of $[\frac{1}{2}, 1)$.

This is in agreement with the range of the divisor \dot{D} in the case of division, which, along with the similarity in the selection rules between the two operations, can be utilized to share the SRT table between the two units (section 3.3).

4.2.2 Range of \dot{S} in the fused unit

To operate on two numbers of arbitrary scales, A and B , which is the case in general-purpose processors, numbers are typically conveyed as pairs of a normalized fraction⁽¹⁾ \dot{A}/\dot{B} and an exponent A_{exp}/B_{exp} , which are processed separately by the arithmetic unit.

For our purpose, which is finding the square root of two multiplied numbers, recovery of the final exponent requires that the radicand's exponent N_{exp} by an even number:

$$N = A \times B$$

$\dot{N} = \dot{A} \times \dot{B} \quad \quad N_{exp} = A_{exp} + B_{exp}$
--

But what if the sum of the exponents above was an odd number (a case we obviously have no control of)? In this case, one of the two significands (\dot{A} or \dot{B}) has to be shifted left or right, while incrementing (or decrementing) the associated exponent, to obtain the desired even sum.

This implies two potential ranges for one of the two significands: $[\frac{1}{2}, 1)$ or $[\frac{1}{4}, \frac{1}{2})$, depending on whether an *exponent-correcting* shift was needed or not, which leads to an all-case, combined range of $[\frac{1}{4}, 1)$ for that significand.

Multiplying this *loosely-normalized* significand with a normalized one in the range $[\frac{1}{2}, 1)$ will yield a radicand in the range $[\frac{1}{8}, 1)$, which corresponds to an extended square-root range of $[\sqrt{\frac{1}{8}}, 1) \approx [0.35, 1) \approx [0.010110101_2, 1.0_2)$.

This range contains binary fractions with either 0 or 1 in the first fractional bit position, which is an important aspect of the problem⁽²⁾ (Fig. 4.1).

But what if we simply accept this extended range and design the SRT table according

⁽¹⁾ also called the *significand* or the *mantissa*.

⁽²⁾ In fact, we'll be referring to this problem as the *loose bit* problem, where fixes and hardware signals intended to deal with it will also follow this "loose-bit" designation.

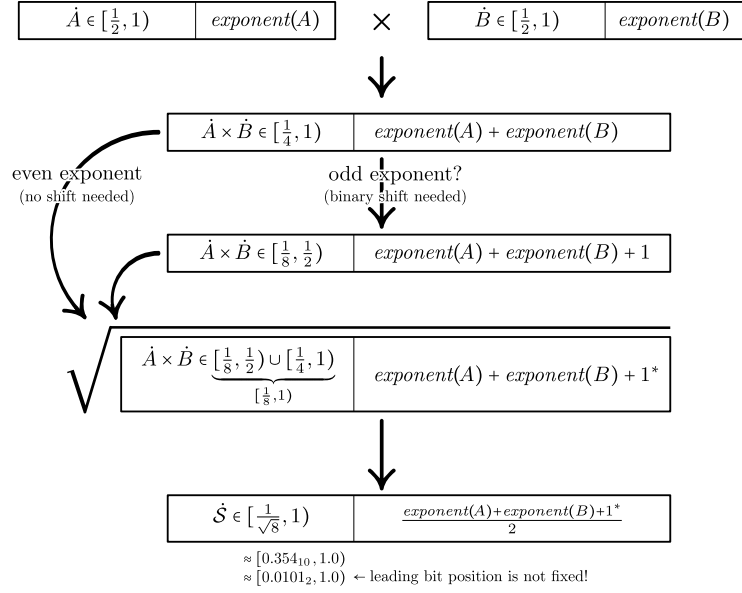


Figure 4.1: An illustration of the argument leading to the wide, problematic range of \hat{S} in a fused multiplication/square-root unit.

to it? This might not seem as a big deal at first, given that for example, an n_S value of three would require only 2 more columns in the table to address the cases when \hat{S} is equal to 0.010_2 or 0.011_2 .

Unfortunately, this will not allow us to exclude the negative digit choices when \hat{S} is exactly equal to $\frac{1}{2}$ (subsection 3.2.5), leading to avoiding the tight overlap regions there in early iteration. It would also introduce even tighter regions in the area between $\hat{S} = \sqrt{\frac{1}{8}} \approx 0.35$ and $\hat{S} = \frac{1}{2}$ (Fig. 4.2).

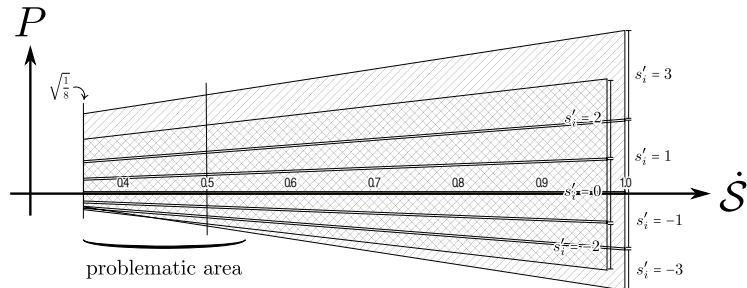


Figure 4.2: The selection regions associated with a maximally-redundant radix-4 system. Note that despite the abundance of overlap, the overlap regions associated with negative digits in the range $\sqrt{\frac{1}{8}} < \hat{S} < \frac{1}{2}$ are incredibly-tight. Note that this diagram was not even plotted for the first (i.e. δ 'th) iteration, in which the irregularity will be severe, but rather for the following one.

In fact, it would be prohibitively-expensive (if not impossible) to adapt to the range

extension in $[\sqrt{\frac{1}{8}}, \frac{1}{2})$, forcing the designer to opt for a modified range of $[\frac{1}{2}, \sqrt{2})$ instead (which results from shifting the loose significand to the left).

This corresponds to an approximate binary range of $[0.1_2, 1.0110101_2) \approx [\frac{1}{2}, 1\frac{1}{2})$ which corresponds to almost doubling the number of columns in the SRT table (compared to the pure square-root case, with a fractional result $\hat{\mathcal{S}}$ in the range $\frac{1}{2}$ to 1).

To fix this problem, we will stick to the $[\sqrt{\frac{1}{8}}, 1)$ range while opting for a combination of the following:

1. An *alternative addressing scheme* of the SRT table when $\hat{\mathcal{S}}_{i-1}$ is in $[\sqrt{\frac{1}{8}}, \frac{1}{2})$, resulting in the same size requirements as the pure square-root case.
2. The design of a *First-Digit Selector*, whose function will be to provide the value of s'_δ as well as determine the leading result bit's position.
3. An increase in the number of required bits by 1 to account for the loose result bit, to ensure a final normalized accuracy of nm significant bits.

In this section, we will discuss the alternative addressing scheme, and the changes it requires in the SRT-table design rules, then we will go through the process of designing the First-Digit Selector.

Note that the last point above will be taken into consideration as part of calculating the total number of iterations in 4.4. For the time being, we will continue to refer to this total count as **iterations**.

4.2.3 Alternative addressing scheme

Using the help of the First-Digit Selector, we will be able to provide the first digit without going into the hassle of adding columns to the SRT table, where otherwise, we will have to modify the table to address the case when $\hat{\mathcal{S}}_{i-1} = 0$.

This has a number of advantages. First, it cancels the need for an initialization scheme where $\hat{\mathcal{S}}$ is assigned the value of 1.0 in the iteration preceding the δ 'th iteration, which is usually done to overcome the limitations in the internal digit set⁽³⁾. This results

⁽³⁾ Due to a less-than-1 $h^{+/-}$ value associated with anything less than a maximally-redundant digit set.

in providing the SRT table with a reliable starting point without adding to its complexity.

Moreover, this will help us achieve a *standardized* SRT-table design process, where otherwise, “gluing” the initialization and the digit-selection problems together would have led to an amount of variation in the design process that is hard to automate.

Second, it will allow us to oversee the case when the the first fractional bit of $\hat{\mathcal{S}}$ is equal to zero, and possibly do something about it, which is the topic of this subsection.

Simply, to eliminate the case when $\hat{\mathcal{S}}$ is less than $\frac{1}{2}$, we are planning on shifting the fractional result $\dot{\mathcal{S}}$ to the left by one bit position prior to passing a truncated copy of it to the SRT table (this implies that one more bit will sometimes be effectively passed to the table, which is how this solution works).

Note that the choice of whether to pass $\dot{\mathcal{S}}$ or $2\dot{\mathcal{S}}$ to the table can be determined by the result’s current first fractional bit (Fig. 4.3).

To examine the effect of passing $2\dot{\mathcal{S}}$ on the table’s design rules, we substitute $2\dot{\mathcal{S}}_{i-1}$ for $\dot{\mathcal{S}}_{i-1}$ in the last correctness constraint (3.6) on page 52:

$$\begin{aligned} 4\dot{\mathcal{S}}_{i-1}(s'_i - h^-) + 2^{-Z}h_B^- &< P_{i-1} < 4\dot{\mathcal{S}}_{i-1}(s'_i + h^+) - 2^{-Z}h_B^+ \\ + 2^Z r^{-i}(s'_i - h^-)^2 && + 2^Z r^{-i}(s'_i + h^+)^2 \end{aligned}$$

Obviously, the constraint has changed, which means that another SRT table designed around these “stretched” selection regions has to be provided, not what we want.

So what we do is that we suggest another variation, which is to complement the act of passing a shifted result $2\dot{\mathcal{S}}_{i-1}$ by passing a shifted copy of the selection remainder $2P_{i-1}$:

$$\begin{aligned} 4\dot{\mathcal{S}}_{i-1}(s'_i - h^-) + 2^{-Z}h_B^- &< \underbrace{2P_{i-1}}_{\text{passed for } P_{i-1}} < 4\dot{\mathcal{S}}_{i-1}(s'_i + h^+) - 2^{-Z}h_B^+ \\ + 2^Z r^{-i}(s'_i - h^-)^2 && + 2^Z r^{-i}(s'_i + h^+)^2 \end{aligned}$$

which, in turn, implies that P_{i-1} would meet:

$$\begin{aligned} 2\dot{\mathcal{S}}_{i-1}(s'_i - h^-) + 2^{-Z-1}h_B^- &< P_{i-1} < 2\dot{\mathcal{S}}_{i-1}(s'_i + h^+) - 2^{-Z-1}h_B^+ \\ + 2^{Z-1}r^{-i}(s'_i - h^-)^2 && + 2^{Z-1}r^{-i}(s'_i + h^+)^2 \end{aligned}$$

which is pretty similar to (but more forgiving than) the original constraint:

$$\frac{2\dot{\mathcal{S}}_{i-1}(s'_i - h^-) + 2^{-Z}h_B^-}{+2^Z r^{-i}(s'_i - h^-)^2} < P_{i-1} < \frac{2\dot{\mathcal{S}}_{i-1}(s'_i + h^+) - 2^{-Z}h_B^+}{+2^Z r^{-i}(s'_i + h^+)^2}$$

So, what we will do is that we will take the maximum of the two lower bounds, and the minimum of the two upper bound, to result in a new constraint that can correctly accommodate a combination of $\dot{\mathcal{S}}_{i-1}/P_{i-1}$ or $2\dot{\mathcal{S}}_{i-1}/2P_{i-1}$ as inputs.

This results in the following, *loose-bit correctness constraint*:

$$\boxed{\frac{2\dot{\mathcal{S}}_{i-1}(s'_i - h^-) + 2^{-Z}h_B^-}{+2^Z r^{-i}(s'_i - h^-)^2} < P_{i-1} < \frac{2\dot{\mathcal{S}}_{i-1}(s'_i + h^+) - 2^{-Z}h_B^+}{+2^{Z-1} r^{-i}(s'_i + h^+)^2}} \quad (4.4)$$

where the only difference is in the factor 2^{Z-1} in the upper bound.

Note that since the difference comes in the iteration-dependent term, it is expected to only affect the first few iterations of the algorithm, which will be helped by our choice to go for a dedicated, first-digit selection mechanism.

Furthermore, as the difference comes in the iteration-dependent term, it is expected not to interfere with our plan to share the SRT table with a fused multiplication/division unit, through evaluating the limits above at an infinite (∞) iteration number.

The good thing about this solution is that it only increases the effective $n_{\mathcal{S}}$ value by one bit, without increasing \hat{P} 's effective bit length, this is because the integral scope of $2P_{i-1}$ when $\dot{\mathcal{S}}_{i-1}$ is within $[\sqrt{\frac{1}{8}}, \frac{1}{2})$ is similar to the scope of P_{i-1} when $\dot{\mathcal{S}}_{i-1}$ is in $[\frac{1}{2}, 1)$.

To avoid ambiguity, we will denote the *normalized truncated result* as $\hat{\mathcal{S}}_{i-1}^*$, which is the amount seen by the SRT table (Fig. 4.3).

4.2.4 Design of the First-Digit Selector

In this subsection, we will start by deriving the correctness criterion associated with the choice of the first digit s'_δ , then from there, we will adopt a simple methodology for the design process, with a design example for a maximally-redundant, radix-4 system.


Design methodology

First, we will choose the number of result bits to be selected in the δ 'th iteration using the First-Digit Selector, which we will denote as m_c .

Note that m_c has to be greater than or equal to 2 bits, in order to reveal the leading bit position of the result, while being less than or equal to m (which is the maximum number of result bits retired per iteration):

$$2 \leq m_c \leq m$$

Note the choice of m_c will influence the Z parameter, which now has to be chosen in such a way as to provide an m_c -bit first digit s'_δ :

$$Z = \delta m - m_c \quad (4.6)$$


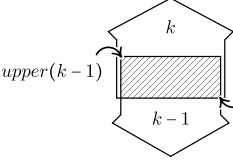
Second, we enumerate all the different values that can be taken by this first digit s'_0 :

$$s'_\delta \in \left\{ \left\lfloor 2^{m_c} \sqrt{\frac{1}{8}} \right\rfloor, 2^{m_c} - 1 \right\}$$

This can be done casually by truncating the binary equivalent of the result's extended range $[0.0101101\cdots_2, 0.1111111\cdots_2]$ to m_c fractional bits, while enumerating the different m_c -bit combinations enclosed between the truncated bounds.

For example, a minimum m_c value of 2 dictates a choice between $\{01_2, 10_2, 11_2\}$ (one, two or three), where the choice cannot be zero, since the result's leading bit is known to be in one of the first two result-bit positions.

After we enumerate all the possible values for the first result digit, we use the criterion (4.5) to obtain the ranges in which a transition can be made from one of the digit choices $k - 1$ to another k , as follows:

$$\begin{aligned}
& \frac{(k - h^-)^2}{+2^{-2Z}r^{+\delta+1}h_B^-} < r^2\mathbf{W}_{i-1} < \frac{(k - 1 + h^+)^2}{-2^{-2Z}r^{+\delta+1}h_B^+}
\end{aligned}
\tag{4.7}$$


These ranges are the equivalent of the overlap regions in the SRT-table design process.

Note that the factor $2^{-2Z}r^{+\delta+1}$ can be simplified to:

$$2^{-2Z}r^{+\delta+1} = 2^{-2Z+m(\delta+1)} = 2^{-Z+(m\delta-Z)+m} = 2^{-Z+m_c+m}$$

To make things neat and easy, we precompute the terms $\pm 2^{-Z+m_c+m}h_B^\mp$ using our knowledge of the Z , m_c and the h_B^\pm parameters. This will result in two fractions, which we might convert to binary by dividing in the binary notation.

Then, we compute $k - h^-$ and $(k - 1) + h^+$ for the different digit values excluding the first value⁽⁴⁾, using our knowledge of h^+ and h^- , which will give us two fractions for each one of the transitional regions.

To work things directly to the point, we square the numbers in the fractions then divide them in binary notation (up to a reasonable number of fractional bits), while adding/subtracting the precomputed terms $\pm 2^{-Z+m_c+m}h_B^\mp$.

This way, we will end up with an approximate lower and upper bounds for each one of the transitional regions.

Then we enumerate all the binary values in-between the bounds while excluding the first one (to abide by the strictly-less-than sign). This will result in columns of ordered bit combinations for each one of the transitional regions.

Note that depending on the internal representation of P , we might also have to drop the second or the last enumerated value as well, to conform to the case of a signed-bit or a carry-save residual, respectively (Fig. 3.8 on page 62).

Then, by looking at the resulting bit sequences for each one of the transitional regions,

⁽⁴⁾ The first digit choice is not associated with a transitional region because there is no valid digit choice lower than it.

we try to find the simplest logical test for each one of the digit choices, using the minimum number of the W register bits.

The resulting logical tests can be implemented directly using combinational logic, which will comprise our First-Digit Selector (Fig. 4.4).



Figure 4.4: The First-Digit Selector. Note that the selector gets its inputs from the W register through hardwired connections.

Design example

In this subsection, we will go through the process of designing a First-Digit Selector for a system that we know has a reasonably-sized SRT table associated with it.

This system is a radix-4 system ($m=2$) that uses a maximally-redundant internal digit set ($h^+ = h^- = 1$) and an irredundant multiplier $b_j \in \{0, 1, 2, 3\}$ ($h_B^- = 0$, $h_B^+ = 1$).

In this design, the Z parameter is given a minimal value of 4 (corresponds to two preliminary iterations, only), while the Selector is to be designed to provide the first full digit of the result ($m_c = 2$).

This dictates a choice between $\{1, 2, 3\}$ for s'_δ , with only two transitional regions: one for the transition from 1 to 2, and another for the transition from 2 to 3.

The 2^{-Z+m_c+m} factor in this design is equal to 1, which signifies a minimal Z choice in this case (more on this later). This means that the $\pm 2^{-Z+m_c+m} h_B^\mp$ terms will have a value of 0 for the lower bound, and -1 for the upper bound.

The design data for the First-Digit Selector is shown in Table 4.1, which assumes an irredundant residual.

Note that if a carry-save residual W was to be supported, then a limited-precision addition has to be performed first, followed by the selection logic.

This will incur a *2ulp* error in the value of W from the selection logic's viewpoint,

where the *ulp* will be that of the adder used, which we will denote as 2^X ⁽⁵⁾.

To adapt to this error, we need to back off from the end of the enumerated range by 2^X , on both sides of Table 4.1. This results in the modified design data of Table 4.2 for a maximal 2^X value of 0001.2 (which corresponds to a 4-bit adder).

Table 4.1: The design data of the First-Digit Selector in a maximally-redundant, radix-4 system, note that the final digit choice is given as two bits: $(s2)(s1)_2$.

transition from 1 to 2		transition from 2 to 3	
$k = 2$		$k = 3$	
$(2 - h^-)^2$	$(1 + h^+)^2$	$(3 - h^-)^2$	$(2 + h^+)^2$
1^2	2^2	2^2	3^2
+ 0	- 1	+ 0	- 1
1	3	4	8
$(0001.)_2$	$(0011.)_2$	$(0100.)_2$	$(1000.)_2$
0001.001		0100.001	
0001.010		0100.010	
:		:	
0001.111		0100.111	
0010.000		0101.000	
:		:	
0010.111		0111.111	
0011.000		1000.000	
ABCD.EFGH		ABCD.EFGH	

choose3 = A
choose2 = $\bar{A}(B + C)$
choose1 = $\bar{A}(\bar{B} + \bar{C})$

s2 = $A + \bar{A}(B + C)$
s1 = $A + \bar{A}(\bar{B} + \bar{C})$

Table 4.2: The design data of the previous Selector for a carry-save residual, and a 4-bit adder.

transition from 1 to 2		transition from 2 to 3	
$k = 2$		$k = 3$	
$(0001.)_2$	$(0011.)_2$ - 2^X	$(0100.)_2$	$(1000.)_2$ - 2^X
$(0001.)_2$	$(0010.)_2$	$(0100.)_2$	$(0111.)_2$
0001.001		0100.001	
0001.010		0100.010	
:		:	
0001.111		0101.000	
0010.000		:	
		0101.111	
		0110.000	
		:	
		0111.000	
ABCD.EFGH		ABCD.EFGH	

choose1 = $\bar{A}\bar{B}\bar{C}$
choose2 = $\bar{A}(B \oplus C)$
choose3 = $A + \bar{A}(BC)$

s2 = $A + \bar{A}(B + C)$
s1 = $A + \bar{A}(\bar{B} \oplus \bar{C})$

⁽⁵⁾ with X being the bit position of the LSB of the limited-precision adder, relative to r^2w_{i-1} .

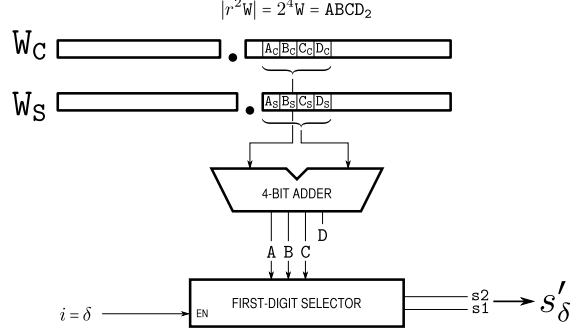


Figure 4.5: The support circuit needed for the First-Digit Selector of Table 4.2.

4.2.5 Analytical minimum bound on Z

Using the inequality in (4.7), we can derive an analytical bound on the value of the Z parameter as a function of the W sample's *ulp*, which we denoted earlier as 2^X .

The logic of this derivation is finding the minimum Z value that secures a height of the transitional region of at least one *ulp*, which corresponds to the minimum Z value permitting the design of a First-Digit Selector.

For this derivation, we will substitute k with the minimum first digit value plus one, since it's the k value associated with the narrowest transitional region:

$$k_{min} \leftarrow \left\lfloor 2^{m_c} \sqrt{\frac{1}{8}} \right\rfloor + 1 \approx 2^{m_c-2} + 1$$

Then, while assuming a symmetric digit set ($h^+ = h^-$), we compute the height of the transitional region as the difference between the two bounds in (4.7):

$$\begin{aligned} & \frac{(k_{min} - 1 + h)^2}{-2^{-Z+m_c+m}h_B^+} - \frac{(k_{min} - h)^2}{+2^{-Z+m_c+m}h_B^-} \\ &= (2k_{min} - 1)(2h - 1) - 2^{-Z+m_c+m}(h_B^+ - h_B^-) \\ &= (2^{m_c-1} + 1)(2h - 1) - 2^{-Z+m_c+m}(h_B^+ - h_B^-) \end{aligned}$$

Then we make up an inequality to ensure that this amount is larger than 2^X :

$$\begin{aligned}(2^{m_c-1} + 1)(2h - 1) - 2^{-Z+m_c+m}(h_B^+ - h_B^-) &> 2^X \\ (2^{m_c-1} + 1)(2h - 1) - 2^X &> 2^{-Z+m_c+m}(h_B^+ - h_B^-)\end{aligned}$$

By finding the binary logarithm of both sides, then rearranging the terms we get:

$$\begin{aligned}Z &> m_c + m + \log_2(h_B^+ - h_B^-) - \log_2((2^{m_c-1} + 1)(2h - 1) - 2^X) \\ Z &> m_c + m + \log_2\left(\frac{h_B^+ - h_B^-}{(2^{m_c-1} + 1)(2h - 1) - 2^X}\right) \\ Z_{min} &= m + m_c + \left\lceil \log_2\left(\frac{h_B^+ - h_B^-}{(2^{m_c-1} + 1)(2h - 1) - 2^X}\right) \right\rceil\end{aligned}\tag{4.8}$$

Note that one thing that is not accounted for in this formula is the restriction put on the value of Z due to the choice of m_c (4.6). For example, this expression returns a value of 3 for the case of our maximally-redundant, radix-4 system above, which is incorrectly related to an m_c value of 2.

That is to say, even though a Z value of 3 is theoretically-sufficient to provide feasible transitional regions, the fact that an m_c value of 2 requires that Z be a multiple of 2 dictates an actual minimum of 4, which is the value chosen previously for our maximally-redundant system.

4.2.6 Aided exclusion of digit choices

Note that the First-Digit Selector can be designed using an h^+ or h^- value smaller than that of the internal digit set. An advantage of this is our ability to deliberately exclude some of the digit choices in the $(\delta+2)$ 'th iteration, which might be advantageous in some cases.

For example, the First-Digit selector of Table 4.2 can be redesigned using an h^- value of $\frac{2}{3}$, while still using an h^+ value of 1. This will have the effect of making the digit choice -3 impossible (or unnecessary to make the result converge) in the $(\delta+2)$ 'th iteration,

which corresponds to the 4'th iteration in this case (leading to the key exclusion of a problematic interval in Fig. 3.16 on page 77).

Henceforth, the design of the First-, as well as Higher-Order Digit Selectors can be used to exclude some of the digit choices in early iterations of the fused algorithm, which otherwise would not be possible in the case of a fused algorithm using dynamic shifting of the $\hat{\mathcal{S}}$ input (subsection 4.2.3).

4.3 The Selection Inputs: P and $\dot{\mathcal{S}}$

To obtain the value of the next result digit s'_i in (4.3), we need to be able to forward the first few bits of P_{i-1} and $\dot{\mathcal{S}}_{i-1}$ to the SRT table, which might involve sampling the W and the S registers at an iteration-dependent position.

In this section, we will work our way through the many layers of abstraction of this thesis in order to relate the SRT table inputs P_{i-1} and $\dot{\mathcal{S}}$ to the actual registers of the algorithm (W and the S), while suggesting hardware solutions for the sampling problem.

4.3.1 Relationship between P_{i-1} and W_{i-1}

The *selection remainder* P_j is defined as follows⁽⁶⁾:

$$P_j \stackrel{\text{def}}{=} r^{-2a} 2^Z r^{j+1} \tilde{R}'_j$$

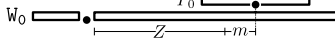
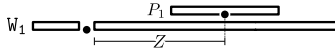


where, to substitute the register value W_j for the remainder \tilde{R}'_j we need to apply⁽⁷⁾:

$$W_j \stackrel{\text{def}}{=} r^{-2a-2j} \tilde{R}'_j \quad \Rightarrow \quad \boxed{\tilde{R}'_j \leftarrow r^{2a-2j} W_j} \quad (4.9)$$

⁽⁶⁾ A combination of $P \stackrel{\text{def}}{=} r^{-2a} 2^Z (rY_j)$ and $Y_j \stackrel{\text{def}}{=} r^j \tilde{R}'_j$ (3.4) (page 50).

⁽⁷⁾ A combination of the residual definition $W_j \stackrel{\text{def}}{=} r^{2j} \tilde{R}'_j$ (2.28) (page 43) and the definition of W_j in the beginnings of this chapter.

Table 4.3

	sampling bit position	
$i = 1$	$-m - Z$	W_0 
$i = 2$	$-Z$	W_1 
$i = 3$	$-Z + m$	W_2 
$i = 4$	$-Z + 2m$	W_3 
	\vdots	

This results in the following relationship:

$$P_j = 2^Z r^{-j+1} W_j$$

$$P_{i-1} = r^{-i} (2^Z r^2 W_{i-1})$$

(4.10)

where the current selection remainder P_{i-1} is the result of sampling W at a changing, or a sliding bit position.

4.3.2 Sampling P from the W register

To deal with the iteration-dependent mapping in (4.10), we compute a sequence of sampling bit positions, for all the iterations from 1 to **iterations**, using:

$$P_{i-1} = 2^Z r^{2-i} W_{i-1} \quad \Rightarrow \quad \boxed{W_{i-1} = 2^{-Z+(i-2)m} P_{i-1}}$$

$$P_{i-1} = 2^{Z+(2-i)m} W_{i-1}$$

This results in the sequence in Table 4.3.

Note that up to the δ 'th iteration of the fused algorithm, these P values won't actually be needed since the result digits will be equal to zero.

As for the δ 'th iteration itself, the digit selection in this iteration is performed by the First-Digit Selector, rather than through the SRT table, which means that the sampling can be restricted to iterations starting from, and following the $\delta + 1$ 'th iteration.

This results in the following, sampling bit-position sequence:

$$\{-Z + (\delta - 1)m, -Z + \delta m, -Z + (\delta + 1)m, \dots, -Z + (\text{iterations} - 1)m\}$$

Note that upon sampling the W register at any one of these bit positions, a range of integral as well as fractional bits needs to be extracted, which will constitute the truncated remainder input \hat{P} of the SRT table.

This involves a sign bit, $n_{P_{integral}}$ integral bits, n_P fractional bits, and an additional fractional bit in case $2P_{i-1}$ was to be passed to the table (subsection 4.2.3).

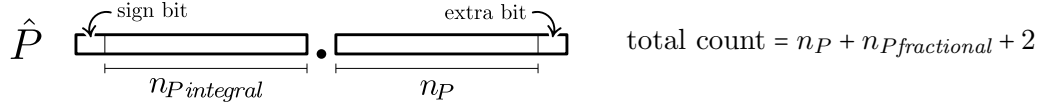


Figure 4.6

Note that only a subset of these bits would be passed to the SRT table. For example, the extra bit is only utilized when $2P_{i-1}$ needs to be passed to the table. Another example is the case of using an unsigned SRT table (subsection 3.2.6), in which the sign bit will not be needed to address the table, but will be used to drive the supporting logic.

To identify this number of bits at a changing bit position in every iteration, we will deploy a shift register with moving 1, which will be called the *iteration indicator* register. At any given time, only one of this register's bits will be equal to one, with the position of that active bit being in tight correspondence with the iteration number i (Fig. 4.7).

Then, we will associate a set of wires with each one of the sampling positions above, while passing the right sample with the help of the iteration-indicator register (Fig. 4.8).

4.3.3 Relationship between \dot{S}_{i-1} and S_{i-1}

In a manner similar to the previous subsection, we start by providing a definition of the fractional result \dot{S}_j , as follows (3.3):

$$\dot{S}_j \stackrel{def}{=} r^{-a} 2^Z S'_j$$

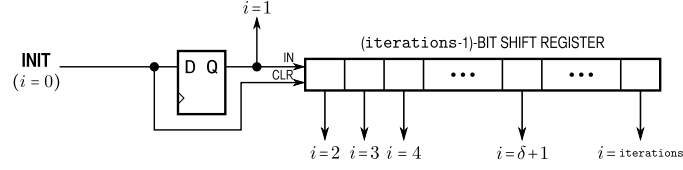


Figure 4.7: Circuit for the iteration-indicator register.

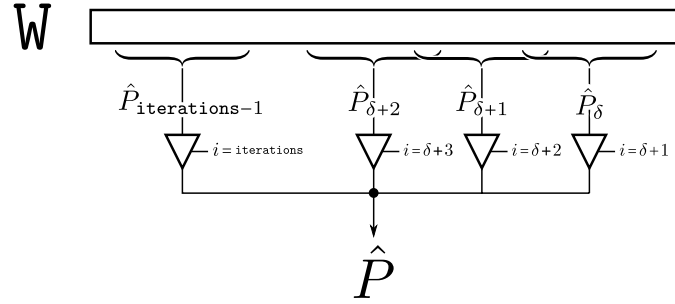


Figure 4.8: Passing the right \hat{P} sample using the help of the iteration-indicator register.

where, to substitute the register value \mathbf{S}_j for the result \mathcal{S}'_j we need to apply⁽⁸⁾:

$$\mathbf{S}_j \stackrel{\text{def}}{=} r^{-(a-j)} \mathcal{S}'_j \quad \Rightarrow \quad \boxed{\mathcal{S}'_j \leftarrow r^{a-j} \mathbf{S}_j}$$

This results in the following relationship:

$$\begin{aligned} \dot{\mathcal{S}}_j &= 2^Z r^{-j} \mathbf{S}_j \\ \boxed{\dot{\mathcal{S}}_{i-1} &= r^{-i} (2^Z r \mathbf{S}_{i-1})} \end{aligned} \tag{4.11}$$

which, similar to the case of P and W , shows that the current fractional result $\dot{\mathcal{S}}_{i-1}$ can be obtained by sampling the \mathbf{S} register at a sliding bit position.

However, it would be a better option in this case to attempt to build a dedicated circuit that stores the value of the first $n_{\mathcal{S}}$ result bits, while taking care of the negative digit values in a simple manner (through carry propagation).

This circuit is shown in Fig. 4.9, which provides the normalized truncated result $\hat{\mathcal{S}}_{i-1}^*$ as its output (the direct input of the SRT table in the fused algorithm).

⁽⁸⁾ A combination of $\mathbf{S}_j = \overline{\mathcal{S}}'_j$ and $\overline{\mathcal{S}}'_j \stackrel{\text{def}}{=} r^{-(a-j)} \mathcal{S}'_j$.

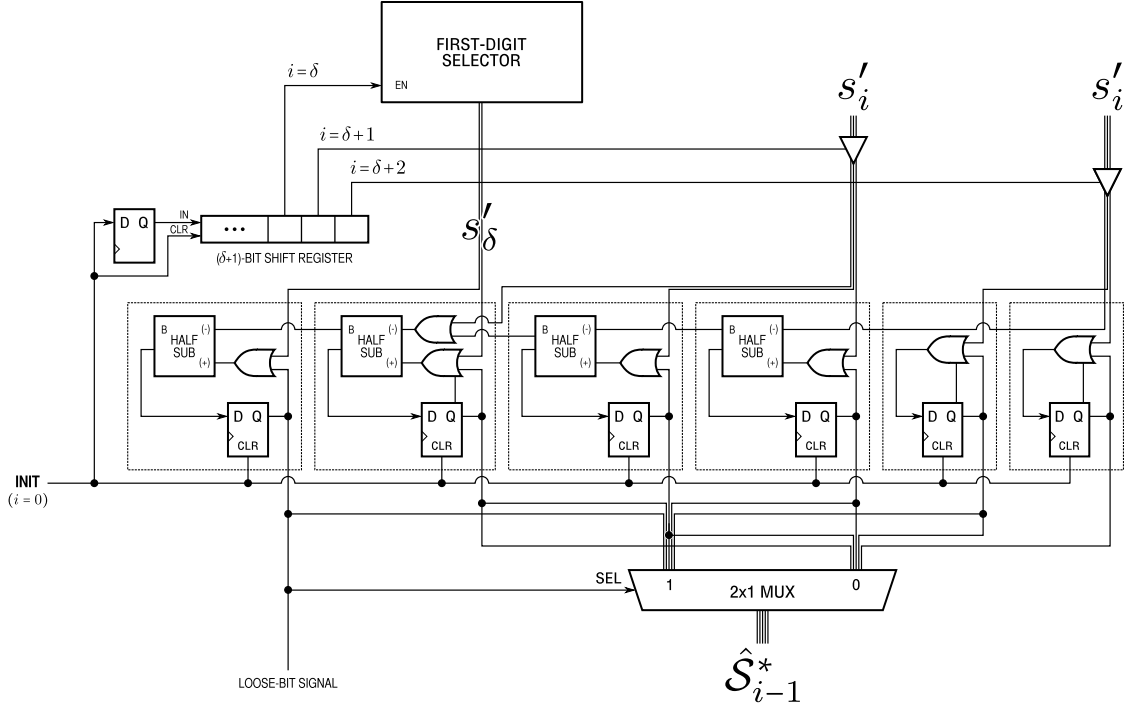


Figure 4.9: The \hat{S} register design for a radix-4 system, an m_c of 2, and an n_S value of 5 (plus an extra bit in case $2\hat{S}$ was to be passed to the table).

4.4 Number of iterations

In this section, we'll discuss the number of iterations needed by the fused algorithm to produce a normalized final result with nm significant bits. To do this, we will come across the topic of the result's final accuracy, while confining the discussion to the simplest mode of accuracy, which is *truncation*.

4.4.1 Final accuracy

In an SRT system, the final accuracy of the result is governed by the internal digit set through the h^+/h^- parameters, as follows:

$$-h^- \text{ulp}(\mathcal{S}_{final}) < \epsilon(\mathcal{S}_{final}) < h^+ \text{ulp}(\mathcal{S}_{final})$$

This implies that the true result, S , is related to \mathcal{S}_{final} in the following way:

$$\mathcal{S}_{final} - h^- ulp(\mathcal{S}_{final}) < \underbrace{\mathcal{S}_{final} + \epsilon(\mathcal{S}_{final})}_{=S} < \mathcal{S}_{final} + h^+ ulp(\mathcal{S}_{final}) \quad (4.12)$$

where in redundant (as opposed to over-redundant) systems, both of h^+ and h^- are less than or equal to one.

4.4.2 Truncated accuracy

The inequality in (4.12) includes two possibilities:

$$\begin{aligned} \mathcal{S}_{final} \leq S < \mathcal{S}_{final} + h^+ ulp(\mathcal{S}_{final}) &\Rightarrow \text{truncate}(S) = \mathcal{S}_{final} \\ \mathcal{S}_{final} > S > \mathcal{S}_{final} - h^- ulp(\mathcal{S}_{final}) &\Rightarrow \text{truncate}(S) = \mathcal{S}_{final} - ulp(\mathcal{S}_{final}) \end{aligned}$$

where the two cases can be distinguished by the sign of the error, $\epsilon(\mathcal{S}_{final})$, which coincides with the sign of the residual.

Hence, to obtain an accurate truncated result, we should be able to detect the sign of the final residual and in the case of a negative sign, decrement the result word.

Note that to reduce the delay of the sign detection in the case of a carry-sum residual, an approximation of the sign can be used, which will incur some error in the final result.

As for decrementing the result, the decremented value can be obtained from the on-the-fly-conversion registers, rather than attempting to decrement \mathbf{S} (subsection 4.5.1).

4.4.3 Rounded accuracy

The issue of rounding will not be discussed in detail in this thesis, rather, we will adhere to the simplistic assumption that rounding requires the production of an extra result bit⁽⁹⁾, which might not be the case if higher precision in rounding was required, or if the full set of the IEEE rounding modes was to be supported [1].

The choice to overlook the issue of rounding is intended to reduce the distraction from the novel contribution of this thesis, which is the incorporation of multiplication.

⁽⁹⁾ called the *rounding bit*.

This is also encouraged by the fact that implementing the different practical, on-the-fly rounding methods would be straightforward [10][22].

4.4.4 Computing the number of iterations

The number of iterations is a function of the total number of result bits that need to be produced by the fused algorithm, which can be broken down as follows:

$$\text{total number of bits} = \underbrace{Z}_{\text{leading zero bits}} + \underbrace{nm}_{\text{actual result bits}} + \underbrace{1}_{\text{loose result bit}} + \underbrace{1}_{\text{rounding bit}}$$

This implies the following total number of iterations:

$$\text{iterations} = n + \left\lceil \frac{Z+2}{m} \right\rceil \quad (4.13)$$

4.5 Hardware Functions

In this section, we will present the partial circuits needed to perform the different functions of the fused algorithm (4.3), which include:

1. Updating the result **S** register,
2. Formation of the $\boxed{\mathbf{S}_{i-1} \mid 0 \mid s'_i}$ -part⁽¹⁰⁾ of the linear-quadratic term,
3. Updating the residual **W** register, and
4. The digit-selection path (using the components and ideas of subsection 4.3).

4.5.1 Updating the result **S** register

In (4.3), the **S** register is updated in a manner that resembles the operation of a shift register, where the old contents are shifted to the left by m bits, followed by the addition of the current result digit s'_i :

$$\mathbf{S}_i = r\mathbf{S}_{i-1} + s'_i$$

⁽¹⁰⁾ also called the **S0s** part.

Namely, if the current result digit was irredundant (i.e. in the range $\{0, \dots, r-1\}$, comprising exactly m bits), the addition would translate to concatenation.

However, as this is known not to be the case in an SRT system, where the result digits s'_i are chosen from a redundant digit set $\phi = \{-\alpha, \dots, +\beta\}$, we are facing the addition of a signed, $(m+1)$ -bit digit that cannot be simply appended to the shifted result $r\mathbf{S}_{i-1}$.

A classical solution to this problem is known in the literature by the name *on-the-fly conversion of digits* [9], which revolves around maintaining two registers instead of one, with one of them storing the value immediately preceding that of the current result (this register is typically denoted as \mathbf{S}_m or \mathbf{S}^-).

To update the \mathbf{S} register, the shifted part is substituted with either $r\mathbf{S}_{i-1}$ or $r\mathbf{S}_{i-1}^-$, depending on the sign of the digit, then, the truncated value of the digit is appended (which corresponds to s'_i or $r-|s'_i|$ depending on whether the digit was positive or negative prior to truncating it).

The process can be simply understood as using the sign bit of the digit to decrement the result (through forwarding \mathbf{S}^- in place of \mathbf{S}), then appending the truncated digit⁽¹¹⁾.

The process for maintaining the \mathbf{S}^- register is similar, with the only difference being that $s'_i - 1$ is used to update the register instead of s'_i .

Namely, the shifted part is substituted with either $r\mathbf{S}_{i-1}$ or $r\mathbf{S}_{i-1}^-$ depending on the sign of $s'_i - 1$, while appending the truncated value of $s'_i - 1$ to the shifted part.

This results in the simple conversion scheme of Table 4.4. Note that this scheme will work with a maximally-redundant digit set, since representing \mathbf{S}^- for the minimum digit value of $-r+1$ implies adding $-r$, which is an $(m+1)$ -bit amount.

This conversion method can be implemented using the simple circuit in Fig. 4.10.

Note that in addition to the role played by the \mathbf{S}^- register in such conversion, it can also be utilized in the case of a negative final residual to avoid decrementing the result word, by delivering \mathbf{S}^- instead of \mathbf{S} at the end of the algorithm (subsection 4.4.2).

⁽¹¹⁾ This only works for negative digits in the 2's complement format. Note that to handle digits in the sign-and-amplitude format (such as those returned by an unsigned SRT table), we will assume that negative digits are converted to the 2's complement format first.

Table 4.4: Simple on-the-fly conversion of digits.

	S^-	S
$s'_i = +\beta$	\uparrow use S	\uparrow use S
$s'_i = +1$	$\boxed{S_{i-1} \mid (s'_i - 1)_m}$	
$s'_i = 0$	$\boxed{S_{i-1}^- \mid (s'_i - 1)_m}$	$\boxed{S_{i-1} \mid (s'_i)_m}$
$s'_i = -1$	\downarrow use S^-	$\boxed{S_{i-1}^- \mid (s'_i)_m}$
$s'_i = -\alpha$		\downarrow use S^-

4.5.2 Formation of the $S0s$ part

Another incident of seemingly “appending” the current result digit s'_i in (4.3) is the case of forming the $S0s$ -part of the linear-quadratic term:

$$W_i = r^2 W_{i-1} - s'_i \times \underbrace{\boxed{S_{i-1} \mid 0 \mid s'_i}}_{\text{the } S0s \text{ part}} + b_{i+1} A_{i-1}^- \times r^{-n} 2^{-2Z}$$

Unlike what the notation suggests, s'_i is a signed $(m+1)$ -bit number that cannot be simply concatenated with $2S_{i-1}$, rather, the whole part has to be formed *on-the-fly* in a

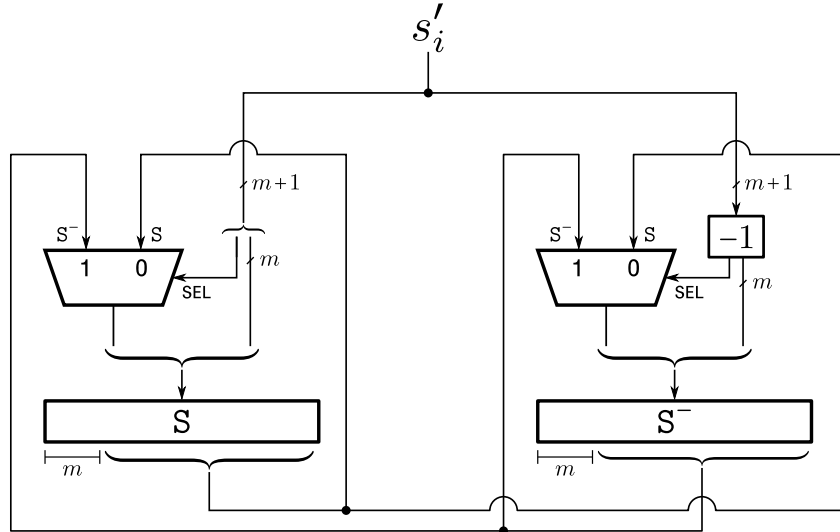


Figure 4.10: On-the-fly conversion of result digits.

manner similar to updating the S register in subsection 4.5.1.

This results in the circuit in Fig. 4.11, which uses the value of the S/S^- registers.

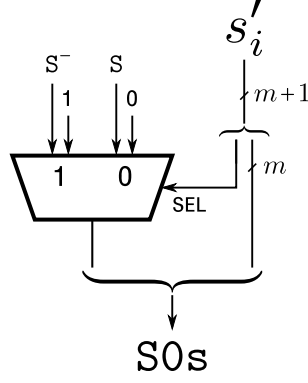


Figure 4.11: On-the-fly formation of the $S0s$ part of the linear-quadratic term.

4.5.3 Updating the residual W register

In (4.3), the W register is updated through the combined act of adding the partial-product term, and subtracting the linear-quadratic term:

$$W_i = r^2 W_{i-1} - \underbrace{s'_i \times \boxed{S_{i-1} \mid 0 \mid s'_i}}_{\text{linear-quadratic term}} + \underbrace{b_{i+1} A_{i-1}^{\leftarrow} \times r^{-n} 2^{-2Z}}_{\text{partial-product term}}$$

Unlike the linear-quadratic term (whose computation requires the value of s'_i), the partial-product term can be computed and added early in the iteration, which will help optimize the critical path of the algorithm.

For this reason, we choose to separate the act of adding the partial-product term from that of subtracting the linear-quadratic term, while using two processors of different lengths for each one of the two tasks.

Note that in the circuits of this section, we will assume a maximally-redundant, radix-4 system with an irredundant multiplier B ($b_j \in \{0, 1, 2, 3\}$).

Adding the partial-product term

To provide the multiplier digits in order, starting from the most-significant, we will be using the register design in Fig. 4.12, where the **LOAD** input will be connected to the initialization signal $i = 0$ while the value of B is fed to the register in parallel.

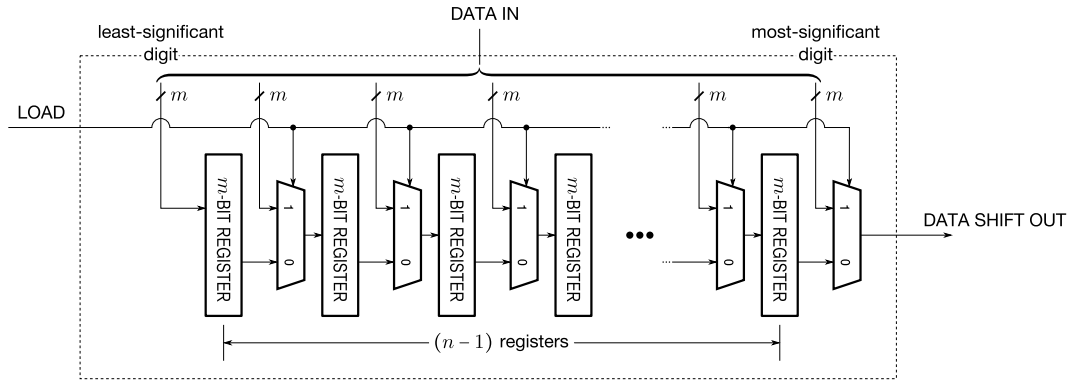


Figure 4.12: The design of the B^{\rightarrow} register.

After that, the digits of the multiplier will be provided through the **DATA SHIFT OUT** output of this register, starting from b_1 (the most-significant digit) in the initialization step ($i = 0$), and up to b_n (the least-significant digit) in the $(n - 1)$ 'th iteration⁽¹²⁾.

Using the digit value b_{i+1} provided through this register, computing the partial product term would be straightforward for an irredundant multiplier B , which is the case of the system assumed in this section (Fig. 4.13).

Note that to figure out the length as well as the correct alignment of the operands (represented by the dots in $W_{S/C}[\dots]$), we need to revisit the register charts in subsection 4.1.1, and namely those of the W and the A^{\leftarrow} registers.

This results in the partial-product-term chart in Fig. 4.14, which can be consulted upon designing the processor used for the addition of the partial-product term.

Subtracting the linear-quadratic term

For the subtraction of the linear-quadratic term, on the other hand, the alignment is easy since such term is purely integral, and hence will only target the integral bits of

⁽¹²⁾ After that, zero-valued digits will be delivered until the end of the algorithm.

the W register, implying a reduced-width processor (Fig. 4.15).

Note that the differences in the case of subtracting the linear-quadratic term are:

1. The subtraction of the linear-quadratic term lies on the critical path of the algorithm (i.e. it has to wait for the selection of current result digit s'_i).
2. The multiplier amount s'_i is signed, expressed in a sign-and-magnitude format⁽¹³⁾.
3. Hence, the 4-to-2 compressor (in the case of our radix-4 system) has to support addition as well as subtraction over two of its 4 inputs, while controlling the function through the digit's *sign* wire.
4. Unlike the addition of the partial product term, the width of the compressor is almost half that of the W registers, with only the uppermost integral bits being involved in the addition/subtraction.

These differences, along with the fact that this subtraction will operate on the output of the previous addition ($r^2W'_S/r^22W'_C$), results in the circuit of Fig. 4.16.

⁽¹³⁾ Note that to facilitate addition/subtraction, we will assume that the digit is converted to this format, even when it's not the native format returned by the SRT table.

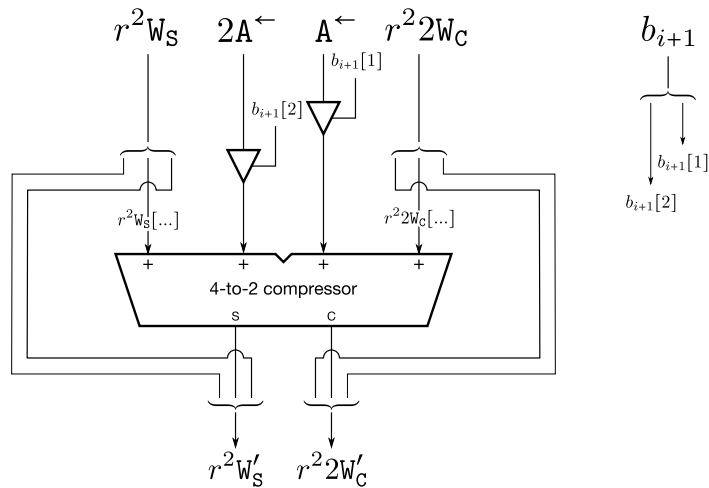
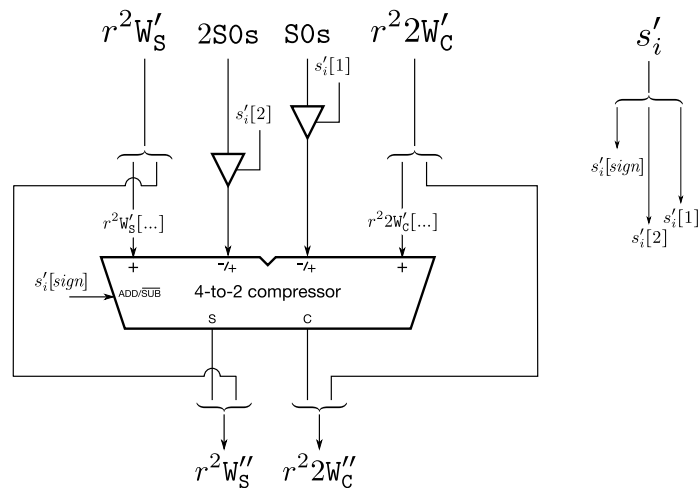
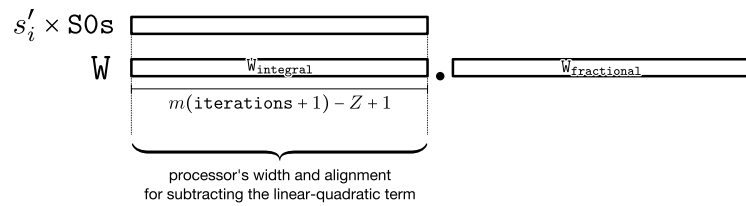
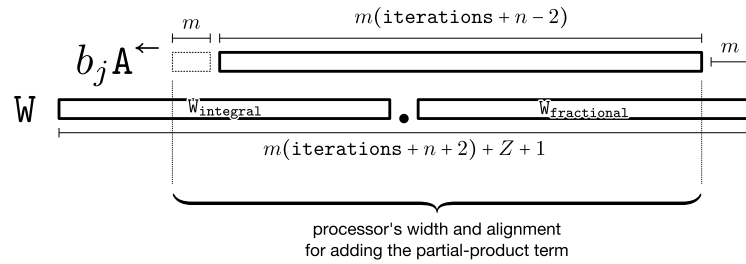


Figure 4.13: Circuit for adding the partial-product term.



4.5.4 The digit-selection path

To combine this information together, we present the circuit in Fig. 4.17, which is intended to support the SRT table design arrived at in the end of the previous chapter.

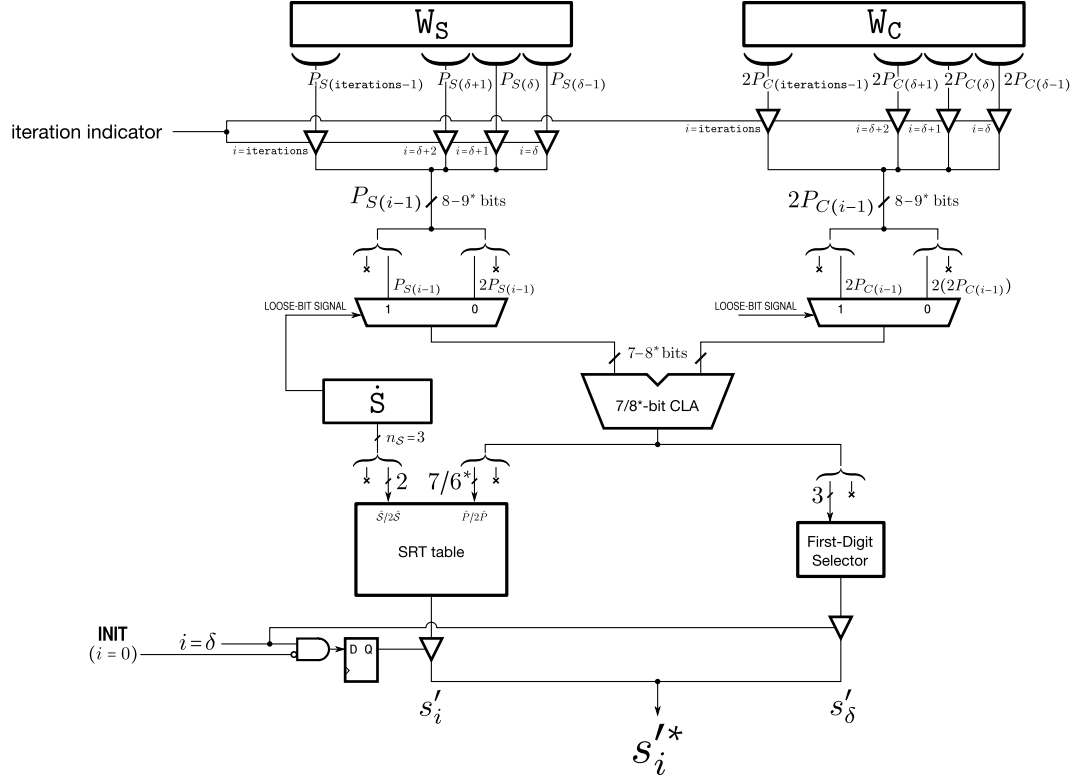


Figure 4.17: The digit selection path for our maximally-redundant radix-4 system, with the choice between two different CLA-adder sizes, resulting in one less or more bit being passed to the SRT table.

remainder \hat{P} , with the higher number corresponding to half-sized table (through the use of the smart constants).

Note that by opting for an unsigned SRT table, which is also possible in this case, we can further reduce the number of P bits used in addressing the table to 5 bits only, which requires an 8-bit CLA adder and some additional support circuitry similar to the one in Fig. 3.12 on page 71 (not shown in the figure).

Note also that we used the same adder for producing the W sample for the First-Digit Selector, as opposed to the dedicated 4-bit adder suggested in Fig. 4.5 on page 97.

The multiplexing logic at the lowermost stage of Fig. 3.12 is designed to return one of the following: (a) zero for iterations preceding the δ 'th iteration, (b) the value returned by the First-Digit Selector for the δ 'th iteration, or (c) the digit value returned by the SRT table for all the following iterations.

4.6 Optimizing the Critical Path

At this point, we are ready for putting together the whole circuit needed for implementing the algorithm in (4.3).

To do this, we will stick to our maximally-redundant, radix-4 system with a value for Z of 4, a simple First-Digit Selector, an n_P value of 4 fractional bits, and an n_S value of 3 fractional bits.

Note that unlike Fig. 4.17, we will stick to an 8-bit sampling CLA adder, while opting for an unsigned SRT table. This will allow us to use a compact SRT table with a total number of input bits of $2 + 5 = 7$ bits.

Hence, the algorithm requires a complex support circuitry, with large registers, and two 4-to-2 compressors. There is also a lot of wiring involved in sampling P , and many stages in the digit-selection path, but the resulting SRT table is compact.

As noted earlier in subsection 4.5.3, the addition of the partial-product term can be moved out of the critical path by performing this addition in parallel with the digit selection, which will require two processors (Fig. 4.18).

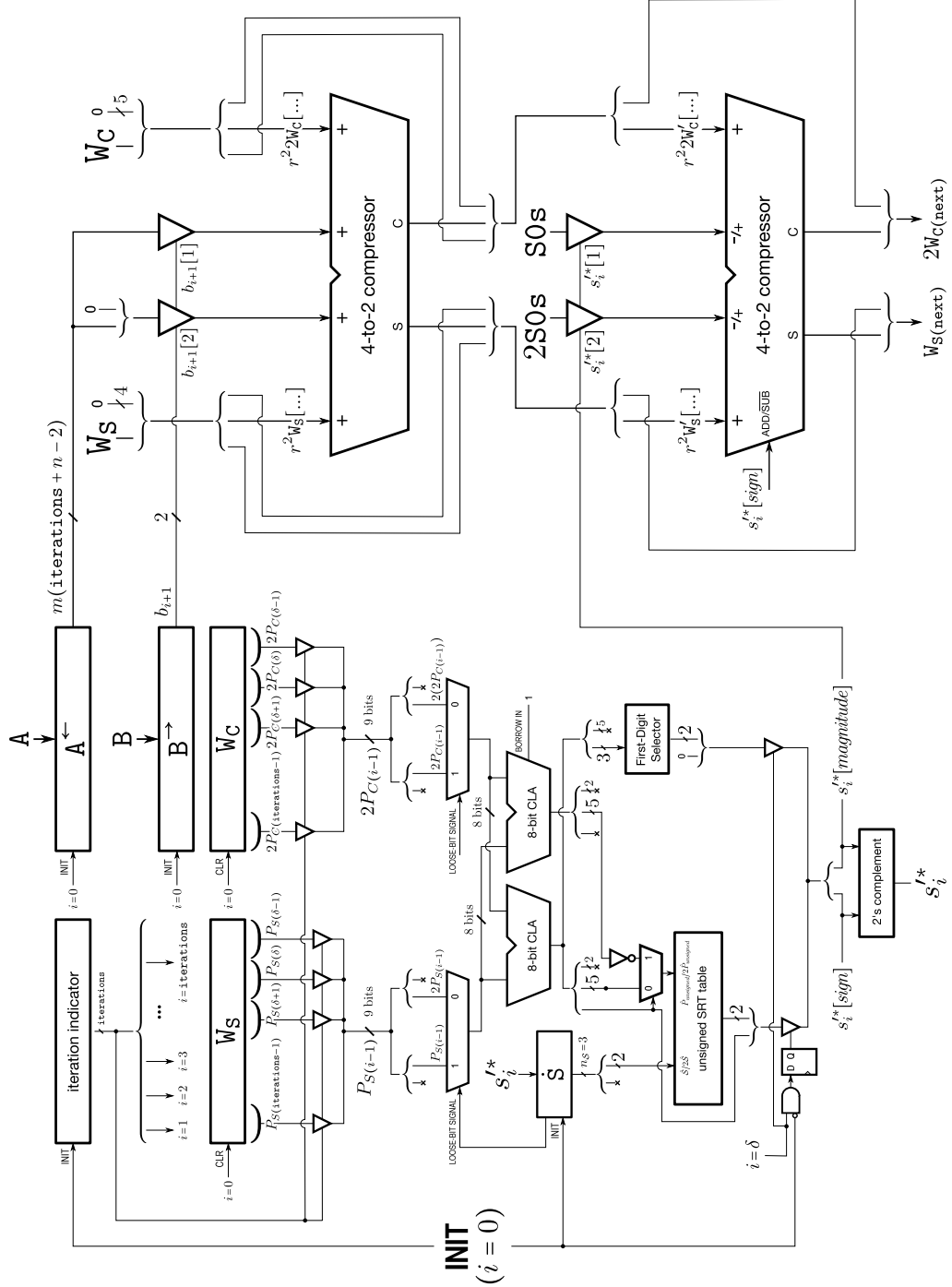


Figure 4.18: The final circuit for the maximally-redundant, radix-4 system. Note that to keep complexity under control, the circuit for on-the-fly conversion is omitted, and so is the circuit for forming the S_0 s part. As for the internal circuits of the iteration-indicator register, the B register, and the S register, they can be easily found in Fig. 4.7, 4.12, and 4.9, respectively. Note that unlike the S register in Fig. 4.9, the one used here is designed for an n_S value of 3, rather than a value of 6.

Chapter 5

Results and Discussion

In this thesis, we went through the theoretical development, the SRT table design, and the hardware design of a fused multiplication/square-root algorithm similar to that of fused multiplication/division in [2].

To test the resulting algorithm as well as the contents of the SRT table, we use the lookup table generator to produce a C-code fragment that summarizes the system parameters. Then, using a special simulation tool, we test whether the algorithm can indeed produce correct results using the table generated by the *Mathematica*[®] code.

The simulation tool is based on software-emulated bit vectors, where all the basic arithmetic and binary operations were rewritten to operate on these bit vectors, which permitted us to watch the execution of the algorithm on a closer level, and to implement automatic detection of overflow, underflow, and other suspicious conditions.

Another advantage of these emulated bit vectors is that we can simulate the tested algorithms to bit lengths beyond those of the machine registers. For example, we can test that the algorithm can still cause the result to converge even after 100, 200, or a 1000 iterations, while producing results as long as 4 thousand bits, which would not be possible if the machine native arithmetic was used to test the algorithm.

Furthermore, the emulated bit vectors allow us to execute different steps of the algorithms as closely as possible to their physical counterparts, resulting in a more meaningful indication of the correctness of the algorithm.

5.1 System Fingerprint

Here is the system fingerprint produced by the lookup table generator for our maximally-redundant, radix-4 system, which only needs to be pasted in the C-language source-code file to allow testing the system⁽¹⁾:

```
// independent system parameters
// - SET 1: BASIC-THEORETICAL
const unsigned short
    algorithm_m = 2,
    algorithm_n = 10,
    algorithm_Z = 4;

// - SET 2: BASIC-PRACTICAL
const unsigned short
    algorithm_alpha = 3,
    algorithm_beta = 3,
    algorithm_ns = 3,
    algorithm_np = 5,
    algorithm_np_fractional = 2,
    algorithm_table_unsigned = 1;

const short SRT_table[] [5] = {
    {4, 4, 4, 3},
    {4, 4, 4, 3},
    {4, 4, 4, 3},
    {4, 4, 4, 3},
    {4, 4, 3, 3},
    {4, 4, 3, 3},
    {4, 4, 3, 3},
    {4, 4, 3, 3},
    {4, 3, 3, 3},
    {4, 3, 3, 3},
    {4, 3, 3, 3},
    {4, 3, 3, 3},
    {4, 3, 3, 3},
    {3, 3, 3, 3},
    {3, 3, 3, 2},
    {3, 3, 3, 2},
    {3, 3, 3, 2},
    {3, 3, 3, 2},
    {3, 3, 2, 2},
    {3, 3, 2, 2},
    {3, 2, 2, 2},
    {3, 2, 2, 2},
    {3, 2, 2, 1},
    {2, 2, 2, 1},
    {2, 2, 2, 1},
    {2, 2, 2, 1},
    {2, 1, 1, 1},
    {2, 1, 1, 1},
    {1, 1, 1, 1},
    {1, 1, 1, 1},
    {1, 0, 0, 0},
    {1, 0, 0, 0},
    {0, 0, 0, 0},
    {0, 0, 0, 0}
};

const unsigned short SRT_table_dimensions[] = {33,4};
const unsigned short SRT_table_p0 = 8 << algorithm_np_fractional;

const unsigned short
    SRT_table_mappings[] [3] = {{0,0,0}};
const unsigned short
    SRT_table_mappings_count = 0;
```

5.1.1 Simulation Output

Once pasted in the simulator, random problems can be solved while maintaining two residual and two result registers: one that is updated using the digits returned by the

⁽¹⁾ In fact, this is not entirely true since special code has also to be written for the emulation of the First-Digit Selector. This is done by emulating the logic in Table 4.2 on page 96.

SRT table, and another that is updated using the exact result digits, which are obtained through a simple, radix-2 algorithm.

This way, by comparing the development of both register sets, we can make sure that things are going the way they should and in case they did not, we can spot the iteration at which the actual result diverged from the theoretical one (indicating an erroneous digit-selection entry in the table).

Below is the simulation output for a random problem, with 10 iterations and a 48-bit result, showing agreement between the theoretical and the actual results:

```
-----
ORWA-AMIN MULTIPLICATIVE SQUARE-ROOT ALGORITHM SIMULATOR
-----
Configuration: <ADVANCED>
- Signed-digit result using on-the-fly digit conversion: YES
- Supports rounding: YES
- Actual digit selection: YES
- Digit selection using an actual table: YES
- Carry-Save residual: YES
- Signed-digit multiplier B with automatic conversion: NO
-----
System parameters:
- m: 2 bits          RADIX = 4
- n: 24 iterations
- Z: 4 bits
-----
System variables:
- size of operands: 48 bits
- iterations: 27 iteration(s)
-----
TYPE OF SIMULATION: One-problem demonstration S = SQRT(A * B)
-----
Problem data:
- A : 201123211030233033210010 (size = 48 bits, radix = 4)
- B : 201021200013330111200221 (size = 48 bits, radix = 4)
- S : 201102201123302212223122 (size = 48 bits, radix = 4)
- S': 00201102201123302212223122 (size = 52 bits)

iteration 0 (initialization):
{S} = 00000000000000000000000000000000
{A} = 00000000000000000000000000000000201123211030233033210010
      (overflow = NO, underflow = NO)
{W} = 0000000000000000000000000000000010023130221211321330200200
      (overflow = NO, underflow = NO)

iteration 1 (b = 0):
000000000000000000000000000000001002313022121132133020020
(^) : 3332
-----
P = "0000" ( 0000)
S = "0"."00"

s' = 0

000000000000000000000000000000001002313022121132133020020
(+) : -----
{W} = 00000000000000000000000000000000100231302212113213302002000
      (overflow = NO, underflow = NO)
{W}t= 00000000000000000000000000000000100231302212113213302002000
      (overflow = NO, underflow = NO)

{S}ac = 00000000000000000000000000000000
{S}th = 00000000000000000000000000000000
{A} = 000000000000000000000000000000002011232110302330332100100

iteration 2 (b = 1):
0000000000000000000000000000000010023130221211321330200200
(^) : 3332
-----
P = "0002" ( 0002)

S. = "0"."00"

S. = "0"

s' = 0

000000000000000000000000000000001002313022121132133020020
(+) : -----
{W} = 00000000000000000000000000000000100231302212113213302002000
      (overflow = NO, underflow = NO)
{W}t= 00000000000000000000000000000000100231302212113213302002000
      (overflow = NO, underflow = NO)

{S}ac = 00000000000000000000000000000000
{S}th = 00000000000000000000000000000000
{A} = 000000000000000000000000000000002011232110302330332100100

iteration 3 (b = 0, s'<precomputed> = 2):
0000000000000000000000000000000010103303202321011300121201000
(^) : 3332
-----
P = "0020" ( 0020)
S. = "0"."00"

s' = FIRST-DIGIT-SELECTOR(ABC = 010) = "2"
0000000000000000000000000000000010103303202321011300121201000
(+) : -----
      1
(-) : -----
{W} = 000000000000000000000000000000001010330320232101130012120100000
{W} = 0000000000000000000000000000000010330320232101130012120100000
      (overflow = NO, underflow = NO)
{W}t= 0000000000000000000000000000000010330320232101130012120100000
      (overflow = NO, underflow = NO)

{S}ac = 00000000000000000000000000000002
{S}th = 00000000000000000000000000000002
{A} = 00000000000000000000000000000000201123211030233033210010000

iteration 4 (b = 2, s'<precomputed> = 0):
0000000000000000000000000000000010330320232101130012120100000
(^) : 3332
-----
P = "0002" ( 0002)
S. = "0"."20"

loose-bit signal = 1

s' = SRTLookUp[31][0] = 0

0000000000000000000000000000000010330320232101130012120100000
(+) : 100231302212113213302002
(-) : -----
{W} = 0000000000000000000000000000000011031212200313303232022102000000
{W} = 0000000000000000000000000000000011031212200313303232022102000000
      (overflow = NO, underflow = NO)
{W}t= 000000000000000000000000000000001103121220031330323202210200000
      (overflow = NO, underflow = NO)
```

```

{S}ac = 000000000000000000000000
{S}th = 000000000000000000000000
{A} = 0000000000000000000000002011232110302330332100100000

iteration 5 (b = 1, s'<precomputed> = 1):
000000000000000000000000000000001103121220031330323202210200000
(^) : 3332
-----
P = "0022" ( 0022)
S. = "0"."20"

loose-bit signal = 1

s' = SRTLookUp[27][0] = 1

000000000000000000000000000000001103121220031330323202210200000
(+) : 20112321103023303321001
(-) : 1001
-----
{W} = 00000000000000000000000000000000110332300330302122230202021000000
{W} = 0000000000000000000000000000000010232300330302122230202021000000
(overflow = NO, underflow = NO)

{W}t= 0000000000000000000000000000000010232300330302122230202021000000
(overflow = NO, underflow = NO)

{S}ac = 00000000000000000000000000000000
{S}th = 00000000000000000000000000000000
{A} = 0000000000000000000000000000000020112321103023303321001000000

iteration 6 (b = 2, s'<precomputed> = 1):
0000000000000000000000000000000010232300330302122230202021000000
(^) : 3332
-----
P = "0021" ( 0021)
S. = "0"."20"

loose-bit signal = 1

s' = SRTLookUp[28][0] = 1

0000000000000000000000000000000010232300330302122230202021000000
(+) : 100231302212113213302002
(-) : 10021
-----
{W} = 000000000000000000000000000000001030233012113000021213222120000000
{W} = 0000000000000000000000000000000022133012113000021213222120000000
(overflow = NO, underflow = NO)

{W}t= 0000000000000000000000000000000022133012113000021213222120000000
(overflow = NO, underflow = NO)

{S}ac = 00000000000000000000000000000000
{S}th = 00000000000000000000000000000000
{A} = 00000000000000000000000000000000201123211030233033210010000000

iteration 7 (b = 0, s'<precomputed> = 0):
0000000000000000000000000000000022133012113000021213222120000000
(^) : 3332
-----
P = "0011" ( 0011)
S. = "0"."20"

loose-bit signal = 1

s' = SRTLookUp[30][0] = 1

0000000000000000000000000000000022133012113000021213222120000000
(+) : 100221
(-) : 100221
-----
{W} = 000000000000000000000000000000002213301211300002121322212000000000
{W} = 13333333333333333333333332131201211300002121322212000000000
(overflow = NO, underflow = NO)

{W}t= 000000000000000000000000000000002213301211300002121322212000000000
(overflow = NO, underflow = NO)

{S}ac = 00000000000000000000000000000000
{S}th = 00000000000000000000000000000000
{A} = 000000000000000000000000000000002011232110302330332100100000000

iteration 8 (b = 0, s'<precomputed> = 2):
13333333333333333333333332131201211300002121322212000000000
(^) : 3332
-----
P = "1303" (-0031)
S. = "0"."20"

loose-bit signal = 1

```

```

s' = SRTLookUp[25][0] = -2

13333333333333333333333332131201211300002121322212000000000
(+) :
(-) : 1333333333333333333333333132231
-----
{W} = 13333333333333333333333213120121130000212132221200000000000
{W} = 000000000000000000000000000000002023212113000021213222120000000000
(overflow = NO, underflow = NO)

{W}t= 000000000000000000000000000000002023212113000021213222120000000000
(overflow = NO, underflow = NO)

{S}ac = 00000000000000000000000000000000
{S}th = 00000000000000000000000000000000
{A} = 0000000000000000000000000000000020112321103023303321001000000000

iteration 9 (b = 0, s'<precomputed> = 2):
000000000000000000000000000000002023212113000021213222120000000000
(^) : 3332
-----
P = "0101" ( 0101)
S. = "0"."20"

loose-bit signal = 1

s' = SRTLookUp[24][0] = 2

000000000000000000000000000000002023212113000021213222120000000000
(+) :
(-) : 2011021
-----
{W} = 0000000000000000000000000000000020232121130000212132221200000000000
{W} = 000000000000000000000000000000001123111300002121322212000000000000
(overflow = NO, underflow = NO)

{W}t= 000000000000000000000000000000001123111300002121322212000000000000
(overflow = NO, underflow = NO)

{S}ac = 00000000000000000000000000000000
{S}th = 00000000000000000000000000000000
{A} = 0000000000000000000000000000000020112321103023303321001000000000

iteration 10 (b = 1, s'<precomputed> = 0):
000000000000000000000000000000001123111300002121322212000000000000
(^) : 3332
-----
P = "0002" ( 0002)
S. = "0"."20"

loose-bit signal = 1

s' = SRTLookUp[31][0] = 0

000000000000000000000000000000001123111300002121322212000000000000
(+) : 20112321103023303321001
(-) :
-----
{W} = 000000000000000000000000000000001131123132113030313210100100000000000
{W} = 000000000000000000000000000000001131123132113030313210100100000000000
(overflow = NO, underflow = NO)

{W}t= 00000000000000000000000000000000113112313211303031321010010000000000
(overflow = NO, underflow = NO)

{S}ac = 00000000000000000000000000000000
{S}th = 00000000000000000000000000000000
{A} = 000000000000000000000000000000002011232110302330332100100000000000

iteration 11 (b = 3, s'<precomputed> = 1):
000000000000000000000000000000001131123132113030313210100100000000000
(^) : 3332
-----
P = "0023" ( 0023)
S. = "0"."20"

loose-bit signal = 1

s' = SRTLookUp[27][0] = 1

000000000000000000000000000000001131123132113030313210100100000000000
(+) : 121010223321203123223003
(-) : 1002211001
-----
{W} = 0000000000000000000000000000000011323333010123030111023301300000000000
{W} = 000000000000000000000000000000001301223000123030111023301300000000000
(overflow = NO, underflow = NO)

{W}t= 000000000000000000000000000000001301223000123030111023301300000000000

```

```

(overflow = NO, underflow = NO)

{S}ac = 0000000000000000201102201
{S}th = 0000000000000000201102201
{A} = 0000000000000020112321103023303321001000000000000

iteration 12 (b = 3, s'<precomputed> = 1):
000000000000000000013012230001230301110233013000000000000
(^) : 3332
-----
P = "0032" ( 0032)
S. = "0"."20"

loose-bit signal = 1

s' = SRTLookUp[25][0] = 2

000000000000000000013012230001230301110233013000000000000
(+) : 121010223321203123223003
(-) : 2011022011
-----
{W} = 000000000000000130303310302230220232213133000000000000
{W} = 13333333333333232011032022302202322131330000000000000
(overflow = NO, underflow = NO)

{W}t= 000000000000000030022210032230220232213133000000000000
(overflow = NO, underflow = NO)

{S}ac = 000000000000002011022012
{S}th = 000000000000002011022011
{A} = 000000000000201123211030233033210010000000000000

iteration 13 (b = 3, s'<precomputed> = 2):
13333333333333332011032022302202322131330000000000000
(^) : 3332
-----
P = "1313" (-0021)
S. = "0"."20"

loose-bit signal = 1

s' = SRTLookUp[27][0] = -1

1333333333333333232011032022302202322131330000000000000
(+) : 121010223321203123223003
(-) : 13333333333333233112233101
-----
{W} = 133333333333332321321023122300121120203330000000000000
{W} = 13333333333333301320321123001211202033300000000000000
(overflow = NO, underflow = NO)

{W}t= 000000000000000333003101022300121120203330000000000000
(overflow = NO, underflow = NO)

{S}ac = 000000000000020110220113
{S}th = 000000000000020110220112
{A} = 000000000000201123211030233033210010000000000000

iteration 14 (b = 0, s'<precomputed> = 3):
133333333333333301320321123001211202033300000000000000
(^) : 3332
-----
P = "1332" (-0002)
S. = "0"."20"

loose-bit signal = 1

s' = SRTLookUp[31][0] = 0

13333333333333333301320321123001211202033300000000000000
(+) :
(-) :
-----
{W} = 133333333333330132032112300121120203330000000000000000
{W} = 133333333333330132032112300121120203330000000000000000
(overflow = NO, underflow = NO)

{W}t= 000000000000003101313301020012112020333000000000000000
(overflow = NO, underflow = NO)

{S}ac = 0000000000000201102201130
{S}th = 0000000000000201102201123
{A} = 00000000000020112321103023303321001000000000000000

iteration 15 (b = 1, s'<precomputed> = 3):
133333333333330132032112300121120203330000000000000000
(^) : 3332
-----
P = "1320" (-0020)
S. = "0"."20"

```

```

loose-bit signal = 1

s' = SRTLookUp[28][0] = -1

1333333333333330132032112300121120203330000000000000000
(+) : 20112321103023303321001
(-) : 13333333333323311223310201
-----
{W} = 1333333333333020011001101103011120203010000000000000000
{W} = 0000000000000022222010021003011120203010000000000000000
(overflow = NO, underflow = NO)

{W}t= 0000000000000022222010021003011120203010000000000000000
(overflow = NO, underflow = NO)

{S}ac = 0000000000002011022011233
{S}th = 0000000000002011022011233
{A} = 0000000000201123211030233033210010000000000000000

iteration 16 (b = 1, s'<precomputed> = 0):
000000000000000022222010021003011120203010000000000000000
(^) : 3332
-----
P = "0011" ( 0011)
S. = "0"."20"

loose-bit signal = 1

s' = SRTLookUp[30][0] = 1

00000000000000222220100210030111202030100000000000000000
(+) : 20112321103023303321001
(-) : 100221100231321
-----
{W} = 0000000000002230212300211210103013001100000000000000000
{W} = 1333333333332202102211013110103013001100000000000000000
(overflow = NO, underflow = NO)

{W}t= 0000000000002230212300211210103013001100000000000000000
(overflow = NO, underflow = NO)

{S}ac = 0000000000020110220112331
{S}th = 0000000000020110220112330
{A} = 0000000002011232110302330332100100000000000000000

iteration 17 (b = 1, s'<precomputed> = 2):
13333333333332202102211013110103013001100000000000000000
(^) : 3332
-----
P = "1310" (-0030)
S. = "0"."20"

loose-bit signal = 1

s' = SRTLookUp[26][0] = -2

13333333333332202102211013110103013001100000000000000000
(+) : 20112321103023303321001
(-) : 13333333333132231132210031
-----
{W} = 1333333333220231000023020100211221111000000000000000000
{W} = 0000000000021333201212323100211221111000000000000000000
(overflow = NO, underflow = NO)

{W}t= 0000000000021333201212323100211221111000000000000000000
(overflow = NO, underflow = NO)

{S}ac = 0000000000201102201123302
{S}th = 0000000000201102201123302
{A} = 0000000020112321103023303321001000000000000000000

iteration 18 (b = 2, s'<precomputed> = 2):
00000000000021333201212323100211221111000000000000000000
(^) : 3332
-----
P = "0103" ( 0103)
S. = "0"."20"

loose-bit signal = 1

s' = SRTLookUp[23][0] = 2

00000000000021333201212323100211221111000000000000000000
(+) : 100231302212113213302002
(-) : 2011022011233021
-----
{W} = 0000000000220032310032103121332113112000000000000000000
{W} = 0000000000012330102302201021332113112000000000000000000
(overflow = NO, underflow = NO)

```

```

{W}t= 00000000001233010230220102133211311200000000000000000
(overflow = NO, underflow = NO)

{S}ac = 0000000002011022011233022
{S}th = 0000000002011022011233022
{A} = 000000020112321103023303321001000000000000000000000

iteration 19 (b = 0, s'<precomputed> = 1):
0000000000012330102302201021332113112000000000000000000000
(^) : 3332
-----
P = "0031" ( 0031)
S. = "0"."20"

loose-bit signal = 1

s' = SRTLookUp[26][0] = 2

0000000000012330102302201021332113112000000000000000000000
(+) :
(-) : 20110220112330221
-----
{W} = 000000000123301023022010213321131120000000000000000000000
{W} = 13333333322132212321020033211311200000000000000000000000
(overflow = NO, underflow = NO)

{W}t= 00000000023013322130023112321131120000000000000000000000
(overflow = NO, underflow = NO)

{S}ac = 0000000020110220112330222
{S}th = 0000000020110220112330221
{A} = 0000002011232110302330332100100000000000000000000000

iteration 20 (b = 0, s'<precomputed> = 2):
1333333332213221232102003321131120000000000000000000000000
(^) : 3332
-----
P = "1310" (-0030)
S. = "0"."20"

loose-bit signal = 1

s' = SRTLookUp[26][0] = -2

13333333322132212321020033211311200000000000000000000000000
(+) :
(-) : 13333333132231132210031121
-----
{W} = 13333333221322123210200332113112000000000000000000000000
{W} = 00000000023031020110322312211311200000000000000000000000
(overflow = NO, underflow = NO)

{W}t= 00000000023031020110322312211311200000000000000000000000
(overflow = NO, underflow = NO)

{S}ac = 0000000201102201123302212
{S}th = 0000000201102201123302212
{A} = 00000201123211030233033210010000000000000000000000000

iteration 21 (b = 2, s'<precomputed> = 2):
00000000023031020110322312211311200000000000000000000000000
(^) : 3332
-----
P = "0112" ( 0112)
S. = "0"."20"

loose-bit signal = 1

s' = SRTLookUp[21][0] = 3

00000000023031020110322312211311200000000000000000000000000
(+) : 100231302212113213302002
(-) : 30132330201212332121
-----
{W} = 00000002310110330121013013330200020000000000000000000000
{W} = 13333333230211303302113201230200020000000000000000000000
(overflow = NO, underflow = NO)

{W}t= 00000000233022312221330232330200020000000000000000000000
(overflow = NO, underflow = NO)

{S}ac = 0000002011022011233022123
{S}th = 0000002011022011233022122
{A} = 00002011232110302330332100100000000000000000000000000

iteration 22 (b = 2, s'<precomputed> = 2):
13333333230211303302113201230200020000000000000000000000000
(^) : 3332
-----

```

```

P = "1312" (-0022)
S. = "0"."20"

loose-bit signal = 1

s' = SRTLookUp[27][0] = -1

13333333230211303302113201230200020000000000000000000000000
(+) : 100231302212113213302002
(-) : 133333233112233102012230221
-----
{W} = 13333323031220121033132111010202200000000000000000000000
{W} = 13333333113330210231303022310202200000000000000000000000
(overflow = NO, underflow = NO)

{W}t= 00000003202100300030020000010202200000000000000000000000
(overflow = NO, underflow = NO)

{S}ac = 0000020110220112330221223
{S}th = 0000020110220112330221222
{A} = 000201123211030233033210010000000000000000000000000000

iteration 23 (b = 1, s'<precomputed> = 3):
13333333113330210231303022310202200000000000000000000000000
(^) : 3332
-----
P = "1322" (-0012)
S. = "0"."20"

loose-bit signal = 1

s' = SRTLookUp[29][0] = -1

13333333113330210231303022310202200000000000000000000000000
(+) : 20112321103023303321001
(-) : 133332331122331020122302221
-----
{W} = 133333120132000102333322010012210000000000000000000000000
{W} = 000000123003003022211013123012210000000000000000000000000
(overflow = NO, underflow = NO)

{W}t= 000000123003003022211013123012210000000000000000000000000
(overflow = NO, underflow = NO)

{S}ac = 0000201102201123302212223
{S}th = 0000201102201123302212223
{A} = 002011232110302330332100100000000000000000000000000000

iteration 24 (s'<precomputed> = 1):
00000012300300302221101312301221000000000000000000000000000
(^) : 3332
-----
P = "0031" ( 0031)
S. = "0"."20"

loose-bit signal = 1

s' = SRTLookUp[26][0] = 2

00000012300300302221101312301221000000000000000000000000000
(-) : 2011022011233022122231
-----
{W} = 13333213002012323022002333122100000000000000000000000000
(overflow = NO, underflow = NO)

{W}t= 000002212130213022331001120221000000000000000000000000000
(overflow = NO, underflow = NO)

{S}ac = 0002011022011233022122232
{S}th = 0002011022011233022122231
{A} = 020112321103023303321001000000000000000000000000000000

iteration 25 (s'<precomputed> = 2):
13333213002012323022002333122100000000000000000000000000000
(^) : 3332
-----
P = "1303" (-0031)
S. = "0"."20"

loose-bit signal = 1

s' = SRTLookUp[25][0] = -2

13333213002012323022002333122100000000000000000000000000000
(-) : 13313223113221003112111021
-----
{W} = 000020110220112330221222312100000000000000000000000000000
(overflow = NO, underflow = NO)

{W}t= 000020110220112330221222312100000000000000000000000000000

```

[illegible]

5.2 Comparisons and Conclusion

By comparing the set of good designs in Table 3.2 against those published in the case of the Multiplier-Divider unit in [2], we can see that through an increased value of Z , the potential use of a Second-Digit Selector, and by benefiting from the smart choice feature, we were able to obtain equivalent truncation pairs in the case of a minimally- and a maximally-redundant radix-4 and radix-8 systems.

Note that the equality of the truncation pairs takes into account the additional integral bit of \hat{P} in the case of the square root (subsection 3.3.1 on page 74).

In the maximally-redundant radix-16 and radix-32 cases, we also seem to be able to provide an equivalent truncation pair at the cost of a higher Z value and the use of a Second-Digit Selector.

In fact, surprisingly, by comparing the total number of input bits in the second row of Table 3.2 to the efficient design in [8], we can see that we were even able to provide the same SRT table size compared to a pure square-root unit of the same radix and internal digit set, with the added advantage of supporting incorporated multiplication.

Note that unlike the circuit in [8], our architecture requires a First-Digit Selector, whose design in the minimally-redundant, radix-4 case is unknown. This First-Digit Selector is not needed in [8].

Still, the conclusion holds, which is that incorporating multiplication, beside adding to the complexity of the data path of the square-root algorithm, seems to only cause a moderate increase (if any) in the size of the SRT table, at the expense of a higher Z value compared to a fused multiplication/division unit.

5.3 Future Work

This work can be improved by exploring the classical route of developing the recurrence relation, which is expected to result in a lower hardware cost.

In fact, the author will produce a scientific paper that will address this shortage, while proposing a minimally-redundant, radix-4 unit based on the alternative recurrence.

Aside from this, more work can be done on standardizing the SRT table design process, and developing the tools presented in this thesis to increase their efficiency and usability. It's also desired to formally work on the issue of SRT-table verification, by creating easy-to-use tools that provide total assurance of the correctness of the contents of the SRT table. Note that there are tools whose purpose is to verify arithmetic circuits, but isolating the problem of verifying of SRT table contents has not been attempted before, and there are no efficient, open-source tools for this.

More work can also be done on modeling and simulating a combined multiplication/-division and a multiplication/square-root unit, and possibly sharing more components than just the SRT table.

Appendix A

Theory of Digit-Recurrence Multiplication/Division*

In this appendix, we will rapidly develop the digit-recurrence multiplication/division theory using the same theoretical approach of Chapter 1, which will demonstrate the power of this approach as well as its educational value.

The development is broken down into two main parts, which are the development of the recurrence relation in A.1, and the derivation of the correctness constrain in A.2.

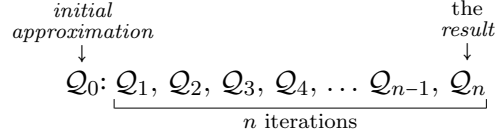
A.1 The Recurrence Relation

Division is defined as finding the number, called the quotient Q , whose multiplication with one of the two operands, named the divisor D , is equal to the other operand, which is named the dividend N :

$$\underbrace{Q}_{\text{quotient}} = \frac{\overbrace{N}^{\text{dividend}}}{\underbrace{D}_{\text{divisor}}} \Rightarrow \boxed{N = Q \times D}$$

The fact that Q is defined indirectly through a criterion dictates that it has to be approached in steps (using search), which implies an iterative algorithm in which an initial approximation Q_0 is repeatedly enhanced through the course of n iterations, to

arrive at a result \mathcal{Q}_n that has an adequate accuracy:



To guide the convergence of the result, we compute the difference between the true dividend N and the dividend implied by the current approximation $\mathcal{N}(\mathcal{Q})$, as follows:

$$\begin{aligned} R &= N - \mathcal{N}(\mathcal{Q}) \\ &= N - \mathcal{Q}D \end{aligned}$$

where this error value is named the remainder R , and is considered as one of the outcomes of division in addition to the result \mathcal{Q}_n itself.

This indirect error value is nothing but a scaled version of the actual error, and hence can be used satisfactorily to direct as well as supervise the development of the result:

$$R = N - \mathcal{Q}D = \mathcal{Q}D - \mathcal{Q}D = \underbrace{(\mathcal{Q} - \mathcal{Q})}_{\text{actual error}} D$$

A.1.1 An additive algorithm

To design an additive algorithm based on maintaining R and \mathcal{Q} , we have to update the remainder additively. This is done by computing the difference between R_i and R_{i-1} :

$$\begin{aligned} R_i &= R_{i-1} + \Delta R_i \\ \Delta R_i &\stackrel{\text{def}}{=} R_i - R_{i-1} \\ &= (N - \mathcal{Q}_i D) - (N - \mathcal{Q}_{i-1} D) \\ &= -(\mathcal{Q}_i - \mathcal{Q}_{i-1}) D \end{aligned} \tag{A.1}$$

In digit-recurrence division, the quotients are enhanced by the addition of a new result digit in every iteration, assuming a most-significant digit position of $a - 1$ and a total length of n digits (equal to the number of iterations). This implies the following

change in the quotient:

$$\mathcal{Q}_i - \mathcal{Q}_{i-1} = \underbrace{q_i}_{\substack{\text{the } i\text{'th} \\ \text{result digit}}} r^{a-i}$$

By substituting this into (A.1) we get:

$$R_i = R_{i-1} - q_i r^{a-i} D$$

A.1.2 Incorporating multiplication

Note that in the case of fused multiplication/division, an approach similar to the one in section 2.4 is used, which includes:

1. Substituting N with the \tilde{N}_i , which is the approximate dividend resulting from accumulating $i + 1$ partial products of $A \times B$ in the i 'th iteration. Note that B is the multiplier, A is the multiplicand, and $A \times B$ is the multiplied dividend in the fused algorithm (i.e. $Q = \frac{A \times B}{D}$).
2. Delaying the actual execution of division by adding Z leading zero bits to Q , which is done by downscaling A by 2^{-Z} (also known as the *processed multiplicand* A').
3. Due to the preprocessing of the multiplicand A , both of the quotients \mathcal{Q}_i and the approximate dividends \tilde{N}_i will be affected as follows:

$\tilde{N}'_i = 2^{-Z} \tilde{N}_i$	$\mathcal{Q}'_i \left(\begin{smallmatrix} \text{with digits} \\ \text{denoted as } q'_j \end{smallmatrix} \right) = 2^{-Z} \mathcal{Q}_i$
-------------------------------------	--

To maintain a remainder based on the approximate processed radicand \tilde{N}'_i we use the same diacritics, namely the *tilde* and the *apostrophe* (to denote approximation and executional delay, respectively), which results in the following notation/definition:

$$\tilde{R}'_i \stackrel{\text{def}}{=} \tilde{N}'_i - \mathcal{Q}'_i D \tag{A.2}$$

where to update this remainder in an additive, digit-based algorithm we need to add:

$$\begin{aligned}
\Delta \tilde{R}'_i &= \tilde{R}'_i - \tilde{R}'_{i-1} \\
&= (\tilde{N}'_i - \mathcal{Q}'_i D) - (\tilde{N}'_{i-1} - \mathcal{Q}'_{i-1} D) \\
&= \underbrace{(\tilde{N}'_i - \tilde{N}'_{i-1})}_{=b_{i+1}Ar^{a-i-1}} - \underbrace{(\mathcal{Q}'_i - \mathcal{Q}'_{i-1})}_{=q'_i r^{a-i}} D
\end{aligned}$$

This implies the following line for updating the remainder:

$$\tilde{R}'_i = \tilde{R}'_{i-1} - q'_i r^{a-i} D + b_{i+1} Ar^{a-i-1} \quad (\text{A.3})$$

To eliminate the dependence on the iteration number i , we counteract the change in position of both terms by shifting the remainder between iterations.

This leads us to introducing the residual W_j as:

$$W_j \stackrel{\text{def}}{=} r^j \tilde{R}'_j \quad \Rightarrow \quad \boxed{\tilde{R}'_j \leftarrow r^{-j} W_j} \quad (\text{A.4})$$

By substituting this into (A.3) we get:

$$\begin{aligned}
r^{\cancel{i}} W_i &= r^{\cancel{i}+1} W_{i-1} - q'_i r^{a-\cancel{i}} D + b_{i+1} Ar^{a-\cancel{i}-1} \\
W_i &= r W_{i-1} - q'_i D r^a + b_{i+1} Ar^{a-1}
\end{aligned}$$

where, by substituting a with either 0 or n for the fractional/integral cases we get the following, fused-operation recurrence relations [2]:

$$\text{fractional case } (a = 0): \quad \boxed{W_i = r W_{i-1} - q'_i D + b_{i+1} Ar^{-1}} \quad (\text{A.5})$$

$$\text{integral case } (a = n): \quad \boxed{W_i = r W_{i-1} - q'_i D r^n + b_{i+1} Ar^{n-1}} \quad (\text{A.6})$$

A.2 The Correctness Constraint

In a manner similar to Chapter 1, we can arrive at the correctness constraint by ensuring that the true (processed) quotient Q' lies in-between, or enclosed by the minimum and the maximum values reachable by future iterations of the algorithm⁽¹⁾, as follows:

$$\begin{array}{ccc}
 \boxed{\mathcal{Q}'_{i-1} \mid q'_i \mid \bar{\alpha} \mid \bar{\alpha} \mid \bar{\alpha} \mid \bar{\alpha} \mid \dots} & & \boxed{\mathcal{Q}'_{i-1} \mid q'_i \mid \beta \mid \beta \mid \beta \mid \beta \mid \dots} \\
 \mathcal{Q}'_{i-1} + (q'_i - h^-)r^{a-i} < & Q' < & \mathcal{Q}'_{i-1} + (q'_i + h^+)r^{a-i} \\
 \mathcal{Q}'_{i-1}D + D(q'_i - h^-)r^{a-i} < & \underbrace{Q'D}_{=N'} < & \mathcal{Q}'_{i-1}D + D(q'_i + h^+)r^{a-i} \\
 D(q'_i - h^-)r^{a-i} < & \underbrace{N' - \mathcal{Q}'_{i-1}D}_{=\tilde{N}'_{i-1} + \epsilon(\tilde{N}'_{i-1})} < & D(q'_i + h^+)r^{a-i} \\
 D(q'_i - h^-)r^{a-i} < & \tilde{R}'_{i-1} + \epsilon(\tilde{N}'_{i-1}) < & D(q'_i + h^+)r^{a-i}
 \end{array}$$

which results in:

$$D(q'_i - h^-)r^{a-i} - \epsilon(\tilde{N}'_{i-1}) < \tilde{R}'_{i-1} < D(q'_i + h^+)r^{a-i} - \epsilon(\tilde{N}'_{i-1}) \quad (\text{A.7})$$

To resolve the error terms, we need to compute the difference between N' and \tilde{N}'_{i-1} :

$$\begin{aligned}
 \tilde{N}'_{i-1} &= \sum_{j=1}^i b_j r^{a-j} A' = \sum_{j=1}^i b_j r^{a-j} 2^{-Z} A \\
 N' &= \sum_{j=1}^n b_j r^{a-j} A' = \sum_{j=1}^n b_j r^{a-j} 2^{-Z} A
 \end{aligned}$$

⁽¹⁾ This corresponds to the repetition of the minimum digit choice $-\alpha$ in the minimum case, or the repetition of the maximum digit value $+\beta$ in the positive case, while assuming a redundant internal digit set of $\phi = \{-\alpha, \dots, +\beta\}$. Note that such a recurring fraction will have a minimum value of $-h^- = -\frac{\alpha}{r-1}$ and a maximum value of $h^+ = \frac{\beta}{r-1}$.

which results in the following value for the error:

$$\begin{aligned}
\epsilon(\tilde{N}'_{i-1}) &= N' - \tilde{N}'_{i-1} \\
&= \sum_{j=i+1}^n b_j r^{a-j} 2^{-Z} A \\
&= r^{a-i} 2^{-Z} A \sum_{j=1}^n b_{i+j} r^{-j}
\end{aligned}$$

noting that, assuming a general digit set of the multiplier B of $\phi_B = \{-B^-, \dots, B^+\}$ and a fractional span of $h_B^- = \frac{B^-}{r-1}$ and $h_B^+ = \frac{B^+}{r-1}$, we get the following range for the sum above:

$$-h_B^- < \sum_{j=1}^n b_{i+j} r^{-j} \approx \sum_{j=1}^{\infty} b_{i+j} r^{-j} < h_B^+$$

This, combined with the following range for an irredundant multiplicand A :

$$0 \leq A < r^a$$

gives us the following minimum and maximum limits for the error terms in (A.7):

$$\begin{aligned}
\epsilon_{min}(\tilde{N}'_{i-1}) &= -r^{2a-i} 2^{-Z} h_B^- \\
\epsilon_{max}(\tilde{N}'_{i-1}) &= +r^{2a-i} 2^{-Z} h_B^+
\end{aligned}$$

By incorporating the minimum error value in the lower bound and the maximum error value in the upper bound of (A.7) to account for the worst-case scenario, we obtain the following:

$$D(q'_i - h^-) r^{a-i} + r^{2a-i} 2^{-Z} h_B^- < \tilde{R}'_{i-1} < D(q'_i + h^+) r^{a-i} - r^{2a-i} 2^{-Z} h_B^+$$

By substituting the recurrence residual W_{i-1} (A.4) for \tilde{R}'_{i-1} in this inequality we get:

$$\begin{aligned} D(q'_i - h^-)r^{a_{\cancel{i}}} + r^{2a_{\cancel{i}}}2^{-Z}h_B^- &< r^{a_{\cancel{i}}+1}W_{i-1} < D(q'_i + h^+)r^{a_{\cancel{i}}} - r^{2a_{\cancel{i}}}2^{-Z}h_B^+ \\ D(q'_i - h^-)r^a + r^{2a}2^{-Z}h_B^- &< rW_{i-1} < D(q'_i + h^+)r^a - r^{2a}2^{-Z}h_B^+ \end{aligned}$$

Then, to standardize the range of the D input, we opt for a normalized fraction form of it \dot{D} (known as the fractional divisor), where:

$$\dot{D} \stackrel{\text{def}}{=} r^{-a}D \quad \Rightarrow \quad \boxed{D \leftarrow r^a \dot{D}}$$

By substituting this in the above then dividing by r^{2a} we get the following, final correctness constraint for combined multiplication and division:

$$\boxed{\dot{D}(q'_i - h^-) + 2^{-Z}h_B^- < \underbrace{r^{-2a}(rW_{i-1})}_{P_{i-1}} < \dot{D}(q'_i + h^+) - 2^{-Z}h_B^+} \quad (\text{A.8})$$

where the amount in the middle is named the selection remainder P , which is the equivalent of the selection remainder in the case of the square root.

The similarity between this correctness constraint and that of fused multiplication/square-root operation is addressed in subsection 3.3, which is used to share the SRT table between the two units.

List of Symbols

m	number of result bits retired per iteration
r	the radix ($= 2^m$)
n	number of radix- r significant digits in the final result
i	the iteration index
ϕ	the result digit set $= \{-\alpha, \dots, \beta\}$
h^-	$= \frac{\alpha}{r-1}$
h^+	$= \frac{\beta}{r-1}$
a	$\begin{cases} 0 & \text{for a fractional result } \bullet \boxed{\leftarrow n \rightarrow} \\ n & \text{for an integral result } \boxed{\leftarrow n \rightarrow} \bullet \end{cases}$
S	true square root
Q	true quotient
A	multiplicand
B	multiplier $= \sum_{j=1}^n b_j r^{a-j}$
D	divisor $= \sum_{j=1}^n d_j r^{a-j}$
ϕ_B	multiplier's digit set $= \{B^-, \dots, B^+\}$
h_B^-	$= \frac{B^-}{r-1}$
h_B^+	$= \frac{B^+}{r-1}$
N	$= A \times B$
S_i	computational square root $= \sum_{j=1}^i s_j r^{a-j}$

$$\begin{aligned}
\mathcal{Q}_i & \quad \text{computational quotient} = \sum_{j=1}^i q_j r^{a-j} \\
\mathcal{S}_{final} & = \sum_{j=1}^n s_j r^{a-j} \\
\mathcal{Q}_{final} & = \sum_{j=1}^n q_j r^{a-j} \\
R_j & = N - \mathcal{S}_j^2 \text{ (square root)} \\
& = N - \mathcal{Q}_j D \text{ (division)} \\
\overline{\mathcal{S}}_j & = \mathcal{S}_j r^{-(a-j)} \\
Z & \quad \text{executorial delay in the fused unit} \\
\delta & = \left\lfloor \frac{Z}{m} \right\rfloor + 1 \\
m_a & = Z - m \left\lfloor \frac{Z}{m} \right\rfloor \\
m_b & = m \left\lceil \frac{Z}{m} \right\rceil - Z \text{ (also called the } excess \text{ bits)} \\
m_c & = m - m_a \\
n^* & = n + \left(\left\lceil \frac{Z}{m} \right\rceil - \left\lfloor \frac{Z}{m} \right\rfloor \right) \\
A' & = 2^{-2Z} A \text{ (square root)} \\
& = 2^{-Z} A \text{ (division)} \\
N' & = 2^{-2Z} N \text{ (square root)} \\
& = 2^{-Z} N \text{ (division)} \\
S'/\mathcal{S}' & = 2^{-Z} S/\mathcal{S} \\
Q'/\mathcal{Q}' & = 2^{-Z} Q/\mathcal{Q} \\
\mathcal{S}'_i & = \sum_{j=1}^i s'_j r^{a-j} \\
\mathcal{Q}'_i & = \sum_{j=1}^i q'_j r^{a-j} \\
\overline{\mathcal{S}}'_j & = \mathcal{S}'_j r^{-(a-j)} \\
\tilde{N}_i & = \sum_{j=1}^{i+1} b_j r^{a-j} A
\end{aligned}$$

\tilde{R}_j	$= \tilde{N}_j - \mathcal{S}_j^2$ (square root)
	$= \tilde{N}_j - \mathcal{Q}_j D$ (division)
\tilde{N}'_i	$= \sum_{j=1}^{i+1} b_j r^{a-j} A'$
\tilde{R}'_j	$= \tilde{N}'_j - \mathcal{S}'_j{}^2$ (square root)
	$= \tilde{N}'_j - \mathcal{Q}'_j D$ (division)
$\epsilon(\tilde{N}_j)$	$= N - \tilde{N}_j$
$\epsilon(\tilde{N}'_j)$	$= N' - \tilde{N}'_j \begin{cases} 2^{-2Z} \epsilon(\tilde{N}_j) & \text{(square root)} \\ 2^{-Z} \epsilon(\tilde{N}_j) & \text{(division)} \end{cases}$
A_j^\leftarrow	shifted multiplicand $= r^j A$
W_j	residual $= r^{2j} \tilde{R}_i$
Y_j	half-pace residual $= r^j \tilde{R}_i$
P_j	selection remainder $= r^{-2a} 2^Z (r Y_j)$
$\dot{\mathcal{S}}_j$	fractional result $= r^{-a} 2^Z \mathcal{S}'_j$
\hat{P}	truncated P_{i-1}
$\hat{\mathcal{S}}$	truncated $\dot{\mathcal{S}}_{i-1}$
n_P	fractional bits in the sample \hat{P}
$n_{\mathcal{S}}$	fractional bits in the sample $\hat{\mathcal{S}}$
$n_{P_{smart}}$	fractional bits that can be passed to the SRT table of \hat{P}
σ	$= \left\lfloor \frac{Z + n_{\mathcal{S}}}{m} \right\rfloor + 1$
2^X	<i>ulp</i> of the adder used to sample $r W_{\delta-1}$ for First-Digit Selection
\dot{D}	fractional divisor $= r^{-a} D$
\hat{P}^*	dynamically-shifted sample, equals to $2\hat{P}$ if $\hat{\mathcal{S}}$ is less than $\frac{1}{2}$, and to \hat{P} otherwise.
$\hat{\mathcal{S}}^*$	dynamically-shifted result, equals to $2\hat{\mathcal{S}}$ if $\hat{\mathcal{S}}$ is less than $\frac{1}{2}$, and to $\hat{\mathcal{S}}$ otherwise.

Bibliography

- [1] IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–58, 29 2008.
- [2] Alaaeldin Amin and M. Waleed Shinwari. High-radix multiplier-dividers: Theory, design, and hardware. *IEEE Transactions on Computers*, 59:1009–1022, 2010.
- [3] D. E. Atkins. Higher-radix division using estimates of the divisor and partial remainders. *IEEE Trans. Comput.*, 17(10):925–934, 1968.
- [4] A. W. Burks, H. H. Goldstine, and J. vonNeumann. *Preliminary Discussion of the Logical Design of Electronic Computing Instrument*. Inst. for Advanced Study, Princeton N. J., 1947–1948.
- [5] Shine Chien Chung. Simplification of lookup table. United States Patent 5,777,917, July 1998.
- [6] Orwa M. Diraneyya. Theory and design of an optimized, iteration-independent look-up table for digit-recurrence division and the square root. Technical report, KFUPM, 2009.
- [7] Orwa M. Diraneyya. Unification of the digit selection function for division and the square root. Technical report, KFUPM, 2009.
- [8] M. D. Ercegovic and T. Lang. Radix-4 square root without initial pla. *IEEE Trans. Comput.*, 39(8):1016–1024, 1990.

- [9] M.D. Ercegovac and T. Lang. On-the-fly conversion of redundant into conventional representations. *IEEE Transactions on Computers*, 36:895–897, 1987.
- [10] M.D. Ercegovac and T. Lang. On-the-fly rounding [computing arithmetic]. *Computers, IEEE Transactions on*, 41(12):1497–1503, dec 1992.
- [11] J. Fandrianto. Algorithm for high speed shared radix 8 division and radix 8 square root. In *Computer Arithmetic, 1989., Proceedings of 9th Symposium on*, pages 68–75, sep 1989.
- [12] C. V. Freiman. Statistical analysis of certain binary division algorithms. In *Proceedings of the IRE*, pages 91–103, January 1961.
- [13] J.B. Gosling and C.M.S. Blakeley. Arithmetic unit with integral division and square root. *Computers and Digital Techniques, IEE Proceedings E*, 134(1):17–23, january 1987.
- [14] Mary Jane Irwin. A pipelined processing unit for on-line division, 1978.
- [15] Peter Kornerup. Revisiting srt quotient digit selection. In *ARITH '03: Proceedings of the 16th IEEE Symposium on Computer Arithmetic (ARITH-16'03)*, page 38, Washington, DC, USA, 2003. IEEE Computer Society.
- [16] Peter Kornerup. Digit selection for srt division and square root. *IEEE Trans. Comput.*, 54(3):294–303, 2005.
- [17] P. Montuschi and L. Ciminiera. On the efficient implementation of higher radix square root algorithms. In *Computer Arithmetic, 1989., Proceedings of 9th Symposium on*, pages 154–161, sep 1989.
- [18] Hooman Nikmehr. *Architectures for Floating-Point Division*. PhD thesis, The University of Adelaide, Australia, August 2005.
- [19] Stuart F. Oberman and Michael J. Flynn. Design issues in floating-point division. Technical report, Stanford University, Stanford, CA, USA, 1994.

- [20] Stuart F. Oberman, Stuart F. Oberman, and Michael J. Flynn. Measuring the complexity of SRT tables. Technical report, Stanford University, Stanford, CA, USA, 1995.
- [21] Behrooz Parhami. Precision requirements for quotient digit selection in high-radix division, 2001.
- [22] Hayward CA Ping Tak Peter Tang. Economical on-the-fly rounding for digit-recurrence algorithms. Patent, 09 2004. US 6792443.
- [23] J. E. Robertson. Selection of quotient digits during digital division. Technical report, University of Illinois, Urbana, 1965.
- [24] James E. Robertson. A new class of digital division methods. *IRE Transactions on Electronic Computers*, pages 88—92, September 1958.
- [25] Ted Williams and Neil Burgess. Choices of operand truncation in the srt division algorithm. *IEEE Transactions on Computers*, 44:933–938, 1995.
- [26] Ted E. Williams and Mark Horowitz. SRT division diagrams and their usage in designing integrated circuits for division. Technical report, Stanford University, Stanford, CA, USA, 1986.
- [27] J. H. P. Zurawski and J. B. Gosling. Design of a hih-speed square root multiply and divide unit. *Computers, IEEE Transactions on*, C-36(1):13 –23, jan. 1987.

Vita

Orwa Mu'men Diraneyya, a Jordanian of a Syrian origin, was born in Amman (Jodan) but grew up in Saudi Arabia. He graduated from high school from the Attarbiya school in Taif, then went for a 5-year BSc program in Computer Engineering in Jordan, where he graduated in 2006 with a GPA of 2.94 (Good) from the Al-Balqa' University for Applied Sciences (BAU) in the city of As-Salt. After that, Orwa came back to Saudi Arabia to enroll in an MSc program in Computer Engineering, which he is about to graduate from today.



Orwa has an interest in Analogue and Digital electronics. He enrolled in optional courses in Analogue electronics in KFUPM and got his undergraduate training in the Electronic Development Center under the Queen Rania's University for Science and Technology in Jordan. His final undergraduate project was the design and implementation of a USB development board, to encourage later students to quit the use of legacy ports (i.e. parallel and serial ports) in their final projects. The board worked, and relied on an ATMEL chip that supported programming through the same USB cable used by the final application, through a jumper on the board.

Orwa is a good writer and a keen programmer, he has written a lot of useful software through his college years and while in KFUPM. He dreams of teaching in the university one day.

Recently, through the help and encouragement of Professor Alaaeldin Amin in the King Fahd University for Petroleum and Minerals (KFUPM), Orwa has developed an interest in the field of Computer Arithmetic, leading him to writing this thesis.