Implementation of binary floating-point arithmetic on embedded integer processors - Polynomial evaluation-based algorithms and certified code generation

Guillaume Revy

N° d'ordre : 549
N° attribué par la bibliothèque : 09ENSL549

ÉCOLE NORMALE SUPÉRIEURE DE LYON
Laboratoire de l'Informatique du Parallélisme

THÈSE

présentée et soutenue publiquement le 1er décembre 2009 par

Guillaume REVY

pour l'obtention du grade de

Docteur de l'Université de Lyon - École Normale Supérieure de Lyon

spécialité : Informatique

au titre de l'École Doctorale Informatique et Mathématiques (InfoMaths) de Lyon

Implementation of binary floating-point arithmetic on embedded integer processors

Polynomial evaluation-based algorithms and certified code generation

Directeur de thèse :	Gilles VILLARD
Co-directeur de thèse :	Claude-Pierre JEANNEROD
Après avis de :	Javier D. BRUGUERA
	Matthieu MARTEL

Devant la commission d'examen formée de :

Christian BERTIN	Membre
Javier D. BRUGUERA	Membre/Rapporteur
Claude-Pierre JEANNEROD	Membre
Matthieu MARTEL	Membre/Rapporteur
Gilles VILLARD	Membre
Paul ZIMMERMANN	Président

À Clémence ...

Remerciements

Avant de mettre un point final à ces trois années de thèse, je souhaiterais remercier toutes les personnes qui ont contribué, directement ou indirectement, à ce qu'elle est aujourd'hui.

En premier lieu, je remercie Paul Zimmermann pour avoir accepté de présider mon jury de soutenance, et pour l'intérêt qu'il a porté à mes travaux. Je remercie également Javier D. Bruguera et Matthieu Martel pour avoir accepté de relire mon manuscrit et pour le soin avec lequel ils l'ont fait. L'ensemble de leurs remarques m'ont permis d'améliorer la qualité de ce manuscrit et je leur en suis très reconnaissant. Je remercie également Christian Bertin d'avoir accepté de faire partie de mon jury. J'apprécie les remarques qu'il a pu me faire lors de ma soutenance.

Je tiens ensuite, et surtout, à remercier mes deux directeurs de thèse, Claude-Pierre Jeannerod et Gilles Villard, pour leur encadrement remarquable et la confiance qu'ils m'ont accordée, depuis mon stage de Master Recherche jusqu'à aujourd'hui. Je remercie Claude-Pierre pour sa disponibilité totale et pour toujours avoir été là pour moi. Merci également pour son suivi permanent et ses conseils toujours pertinents. Nos discussions nombreuses m'ont beaucoup apporté et m'ont permis d'avancer dans mes recherches même dans les moments de doute. Je remercie Gilles qui, malgré la charge qui incombe au directeur du projet Arénaire, puis du LIP, a toujours su être présent au bon moment. Tous deux ont toujours été d'excellent conseil, et leurs encouragements pendant ces trois années m'ont été d'un réel soutien. J'espère un jour pouvoir encadrer mes étudiants aussi bien qu'ils l'ont fait pour moi.

Au cours de ces trois années, j'ai eu la chance de collaborer avec Hervé Knochel et Christophe Monat. Je les remercie pour leur accueil, leur aide et leurs réponses à mes questions (parfois naïves) de compilation... Ça a été un réel plaisir de travailler avec eux.

Ensuite, mes remerciements vont à toutes les personnes que j'ai croisées au LIP, à commencer par les membres (ou anciens membres) du projet Arénaire... merci à Nathalie, Arnaud, Edouard, Florent, Damien, Jean-Michel, Nicolas B., Nicolas L., Vincent et Serge, pour leur aide, leur conseil et leur soutien pendant ces trois années. Mais également merci à Alain (notamment pour m'avoir fait découvrir les joies de la Champions League à Gerland), Daniel, Dom,... ainsi qu'aux assistantes: Caroline, Corinne, Isabelle, Marie et Sylvie pour leur efficacité (et véritablement, pour moi, elles ont été d'une grande aide).

Merci bien évidemment à tous mes co-bureaux qui ont su me supporter pendant ces trois années passées au LIP... merci donc à Nicolas, Francisco, Ivan, Diep, Mioara (et désolé de t'avoir empêchée de parler dans le bureau pendant mes deux derniers mois de rédaction !), Christoph et Andrew. Et de manière générale, merci à tous les doctorants/post-doctorants du projet Arénaire: Jingyan, Guillaume M., Jérémie, Sylvain C., Bogdan, Adrien, Christophe, Erik, Álvaro,... pour leur coup de main, leur conseil, ou tout simplement pour les discussions que j'ai pu avoir avec eux. Bien d'autres méritent d'être remerciés ici... merci à Veronika, Rémi, Seb, Alexandru, Ludo, Romaric, ainsi qu'à tous les doctorants du LIP, avec qui j'ai pu partager une discussion scientifique ou non, au labo ou ailleurs. Plus particulièrement merci à Mathilde, Sylvain S., Damien R. et Manu, pour tous les moments inoubliables passés avec eux, des soirées à Saint-Georges, au concert

de Java au CCO, ou celui de Polémil Bazar sur la péniche du Sirius,... en passant par les simples discussions autour d'un verre ou bien encore les matchs de foot à l'ENS ou à Croix-Rousse... et grâce à qui ces quelques années à Lyon ont été très agréables.

J'en profite pour remercier Christian Vial et Emmanuel Dellandréa, mes tuteurs à l'École Centrale de Lyon, pour leur soutien et leur aide pendant mes trois années de monitorat, ainsi que tous les étudiants du Département Mathématiques et Informatique que j'ai croisés... en espérant leur avoir apporté quelque chose. J'en profite également pour remercier Frédéric Gaffiot, mon maître de stage de DUT à l'École Centrale de Lyon, pour m'avoir toujours soutenu et encouragé dans mon choix de carrière.

J'ai une pensée également pour tous mes amis de Belfort, de Lyon et de mon Jura natal, celles et ceux qui m'ont soutenu et ont réussi à me sortir la tête du guidon à un moment ou à un autre (en organisant des jeux dans Lyon, des week-ends en gîtes dans les Alpes,...). Je ne les nomme pas ici par peur d'en oublier... mais je suis sûr, ils se reconnaîtront.

Avant de conclure, je remercie ma famille, notamment ma sœur et mes parents, pour m'avoir toujours guidé dans la bonne direction, et pour m'avoir toujours soutenu dans ce que je faisais, parfois même sans vraiment savoir où j'allais.

Enfin, je remercie Clémence, qui m'accompagne depuis si longtemps. Merci pour ta patience pendant ces trois années de thèse. Ta présence, particulièrement pendant les derniers mois de rédaction, a été un réel soutien. Enfin, merci d'avoir laissé cette magnifique ville qu'est Lyon (et tout ce qui va avec), pour venir vivre le "rêve californien" avec moi... véritablement, ça aurait été tout de suite moins rock 'n' roll sans toi !

Merci à vous.

Contents

Contents vii

List of figures xi

List of tables xiii

List of algorithms xv

Introduction 1

1 Context 9
1.1 Overview of the ST231 architecture . . . 9
1.1.1 Some features of the ST231 architecture and compiler . . . 9
1.1.2 *If-conversion*: reduction of conditional branches . . . 11
1.1.3 Bundle and instruction encoding . . . 12
1.1.4 Data accesses in memory . . . 16
1.1.5 Elements of the instruction architecture set . . . 16
1.2 Binary floating-point and fixed-point arithmetics . . . 18
1.2.1 Binary floating-point arithmetic . . . 18
1.2.2 Binary interchange format encoding . . . 20
1.2.3 Binary fixed-point arithmetic . . . 22

I Design of optimized algorithms for some binary floating-point operators and their software implementation on embedded VLIW integer processors 27

2 Basic blocks for implementing correctly-rounded floating-point operators 29
2.1 Unpacking an input floating-point number . . . 30
2.1.1 Integer X encoding the binary floating-point number x . . . 30
2.1.2 Sign and biased exponent extraction . . . 30
2.1.3 Normalized significand extraction . . . 33
2.2 Implementation of rounding algorithms . . . 35
2.2.1 Correctly-rounded value of a real number . . . 35
2.2.2 Rounding-direction attributes . . . 36
2.2.3 When the exact result is finite . . . 38
2.2.4 When the exact result may have an infinite number of digits . . . 42
2.3 Packing the correctly-rounded result . . . 47
2.3.1 Explicit formula for the standard encoding of the correctly-rounded result . . . 47
2.3.2 Overflow handling . . . 50

2.4 Example of correctly-rounded multiplication . . . 51
2.4.1 Range reduction . . . 51
2.4.2 Computation of the correctly-rounded significand . . . 52
2.4.3 Implementation of multiplication in the *binary32* format . . . 55

3 A uniform approach for correctly-rounded roots and their reciprocals 59
3.1 General properties of the real function $x \mapsto x^{1/n}$ . . . 60
3.1.1 Basic properties of $x^{1/n}$ in binary floating-point arithmetic . . . 62
3.1.2 Range reduction of $x^{1/n}$ . . . 63
3.1.3 Properties useful for correct rounding . . . 65
3.2 Computation of a one-sided approximation . . . 69
3.2.1 Bivariate polynomial approximation . . . 70
3.2.2 Certified approximation and evaluation error bounds . . . 72
3.2.3 Automatic generation of polynomial coefficients . . . 74
3.2.4 Evaluation of the bivariate polynomial . . . 76
3.3 Sign and exponent result implementation . . . 78
3.3.1 Sign result computation . . . 78
3.3.2 Exponent result computation . . . 79
3.4 Implementation of special input handling . . . 80
3.4.1 Definition of special operands for $x^{1/n}$ . . . 81
3.4.2 How to filter out special input? . . . 81
3.4.3 How to determine the output to be returned? . . . 82
3.5 First application example: square root ($n = 2$) . . . 85
3.5.1 Range reduction for square root . . . 85
3.5.2 Result exponent computation . . . 85
3.5.3 Extraction of the evaluation point $(\hat{s}^*, t^*)$ . . . 86
3.5.4 Bivariate approximant computation for *binary32* implementation . . . 88
3.5.5 Bivariate polynomial evaluation program . . . 90
3.5.6 How to implement the rounding condition? . . . 95
3.6 Second application example: reciprocal ($n = -1$) . . . 98
3.6.1 Range reduction for the reciprocal . . . 99
3.6.2 Result sign and exponent computation . . . 99
3.6.3 Special input, underflow, and overflow . . . 100
3.6.4 How to approximate the exact value ℓ ? . . . 102
3.6.5 Implementation of the rounding condition . . . 104

4 Extension to correctly-rounded division 111
4.1 General properties of division . . . 112
4.2 Result sign and exponent computation . . . 113
4.2.1 Result sign computation . . . 114
4.2.2 Result exponent computation . . . 114
4.3 Special input handling . . . 115
4.3.1 IEEE 754 specification of division . . . 115
4.3.2 Filtering out special inputs . . . 115
4.3.3 Deciding the result to be returned . . . 116
4.4 Correctly-rounded division by digit recurrence . . . 116
4.4.1 General principle . . . 116
4.4.2 Restoring division . . . 117
4.4.3 Nonrestoring division . . . 119
4.4.4 Nonrestoring division on the ST231 processor . . . 122

4.4.5 How to achieve correct rounding? . . . 123
4.4.6 Performances of the (non)restoring algorithms on the ST231 processor . . . 125
4.5 Bivariate polynomial evaluation and validation . . . 126
4.5.1 Division via bivariate polynomial evaluation . . . 126
4.5.2 Example of implementation for the *binary32* format . . . 129
4.5.3 Validation using a dichotomy-based strategy . . . 133
4.6 Rounding condition implementation . . . 135
4.6.1 General definitions . . . 135
4.6.2 Properties useful for correct rounding . . . 136
4.6.3 How to implement the rounding condition? . . . 138

II Code generation for the efficient and certified evaluation of polynomials in fixed-point arithmetic 143

5 Polynomial evaluation in fixed-point arithmetic on the ST231 processor 145
5.1 Classical polynomial evaluation schemes . . . 146
5.1.1 Horner's rule . . . 146
5.1.2 Extension to second-order Horner's rule . . . 151
5.1.3 Estrin's method . . . 155
5.1.4 Numerical results on fixed-point polynomial evaluation on ST231 processor . 159
5.2 Polynomial evaluation via coefficient adaptation . . . 160
5.2.1 What is coefficient adaptation? . . . 161
5.2.2 Knuth and Eve's algorithm . . . 161
5.2.3 Paterson and Stockmeyer's algorithm . . . 163

6 Computing efficient polynomial evaluation programs 165
6.1 Computing all the parenthesizations . . . 167
6.1.1 Preliminary definitions . . . 167
6.1.2 Building rules . . . 168
6.1.3 Description of the algorithm and implementation of the building rules . . . 170
6.1.4 Example: evaluation of general bivariate degree-2 polynomials . . . 172
6.1.5 Improvement for some "special" polynomials . . . 175
6.1.6 Numerical results . . . 177
6.2 Computing parenthesizations of low evaluation latency . . . 179
6.2.1 Definition of the *target latency* . . . 179
6.2.2 Determination of a *dynamic* target latency . . . 180
6.2.3 Numerical results . . . 182
6.3 Optimized search of *best* parenthesizations . . . 186
6.3.1 Recursive search of best splittings of polynomials . . . 186
6.3.2 Numerical results . . . 187
6.4 Generating efficient and certified polynomial evaluation programs . . . 189
6.4.1 Presentation of the general framework . . . 190
6.4.2 Efficient parenthesization selection and certified code generation . . . 191
6.4.3 Last examples of generated programs . . . 192

Conclusion 195

III Appendices **199**

A Notation **201**

B C code for implementing various basic integer operators **203**
B.1 Implementation of max operators 203
B.2 Implementation of min operators 203
B.3 Implementation of $32 \times 32 \rightarrow 32$ multiplications 203
B.4 Implementation of *count leading zeros* operator 204

Bibliography **205**

List of figures

1 General framework for the automatic generation of efficient and certified polynomial evaluation programs. . . . 7

1.1 Block diagram of the ST231 architecture. . . . 10
1.2 The two possible encodings of the `mul64hu` instruction. . . . 13
1.3 Binary interchange format encoding of floating-point data. . . . 20

2.1 Flowchart for correctly-rounded function implementation, for univariate function. . . . 31
2.2 Rounding-direction attributes. . . . 37
2.3 RoundTiesToEven for $r > 0$. . . . 39
2.4 RoundTowardPositive for $r > 0$. . . . 40
2.5 RoundTowardNegative for $r > 0$. . . . 41
2.6 Relationship between $\ell \cdot 2^{-\lambda_r}$ and u, when ℓ is approximated from above. . . . 42
2.7 RoundTiesToEven for $r > 0$. . . . 44
2.8 RoundTowardPositive for $r > 0$. . . . 45
2.9 RoundTowardNegative for $r > 0$. . . . 46
2.10 Rounding, when overflow occurs. . . . 50

3.1 Graph of $x^{1/n}$ for various values of n. . . . 61
3.2 Evaluation tree for square root implementation using *best* scheme. . . . 93

4.1 Flowchart for nonrestoring iteration using `divs` and `addcg` instructions. . . . 123
4.2 Absolute approximation error of $a(t)$ with respect to $1/(1+t)$ over $[0, 1-2^{-23}]$. . . . 131
4.3 Absolute evaluation error $\rho(\mathcal{P})$, for $t \geq 0.9749044179916381835937 5$. . . . 133
4.4 Absolute approximation error $\alpha(a)$, around $t = 0.9749044179916381835937 5$. . . . 134

5.1 Evaluation tree for square root implementation using *second-order Horner's rule*. . . . 154
5.2 Evaluation tree for square root implementation using *Estrin's method*. . . . 157
5.3 Evaluation tree for square root implementation using *second Estrin's method*. . . . 158

6.1 Evaluation tree of polynomial $a(x)$. . . . 166
6.2 Building of a valid expression e or a power of degree k. . . . 169
6.3 Strategy for parallel generation of parenthesizations for evaluating univariate degree-3 polynomials. . . . 177
6.4 Latency for the evaluation of $x^{n_x} = x^k \times x^{n_x - k}$. . . . 181
6.5 Target latency for the evaluation of particular bivariate polynomials of the form $P(s,t) = 2^{-p-1} + s \cdot a(t)$. . . . 185
6.6 Impact of forcing the leading coefficient to be a power of 2. . . . 185
6.7 Optimized search of *best* parenthesizations of univariate polynomial. . . . 187

6.8 General framework for the automatic generation of efficient and certified polynomial evaluation programs. 191

List of tables

1 Performances on ST231 in # cycles [# integer instructions]. . . . 4
2 Speed-up of FLIP 0.3 compared to STlib (FLIP 0.3 vs STlib), of FLIP 1.0 compared to STlib (FLIP 1.0 vs STlib), FLIP 1.0 compared to FLIP 0.3 (FLIP 1.0 vs FLIP 0.3), in RoundTiesToEven. . . . 5

1.1 Elements of the ST231 instruction architecture set. . . . 17
1.2 Basic binary floating-point formats parameters. . . . 18
1.3 Interchange format parameters for the basic binary floating-point formats. . . . 20
1.4 Relationship between floating-point datum x and its encoding into integer X ($k = p + w$). . . . 23

2.1 Relationship between $\ell \cdot 2^{-\lambda_r}$ and m_r, for $\circ \in \{\mathrm{RN}_p, \mathrm{RU}_p, \mathrm{RD}_p, \mathrm{RZ}_p\}$. . . . 37
2.2 Specification of the multiplication operation $x \times y$. . . . 55

3.1 Summary of properties useful for rounding $x^{1/n}$ correctly (for non trivial input). . . . 65
3.2 Evaluation error bound, according to n. . . . 74
3.3 Degree of the polynomial approximant $a(t)$, and approximation of the certified error bounds θ and η, for various values of n, and for $(k, p) = (32, 24)$. . . . 76
3.4 Special values for $x^{1/n}$ for $n \geq 2$. . . . 81
3.5 Special values for $x^{1/n}$ for $n \leq -1$. . . . 81
3.6 Relationship between input x, encoding integer X, and the bit string of the returned value, for $x \in \{\pm 0, +\infty\}$. . . . 84
3.7 Numerical estimation of $\alpha(a^*)$, for $6 \leq \delta \leq 11$. . . . 88
3.8 Coefficients of polynomial approximant for square root implementation. . . . 89
3.9 Coefficients of polynomial approximant for square root implementation. . . . 90
3.10 Performances on ST231, for square root implementation using unstructured polynomial coefficients. . . . 91
3.11 Feasible scheduling on ST231, for square root implementation. . . . 94
3.12 Performances on ST231, for square root implementation using *structured* polynomial coefficients. . . . 95
3.13 Feasible scheduling on ST231, for square root implementation with structured coefficient polynomial. . . . 95
3.14 Special output for x^{-1}. . . . 101
3.15 Numerical estimation of $\alpha(a^*)$, for $6 \leq \delta \leq 11$. . . . 103
3.16 Coefficients of polynomial approximant for reciprocal implementation. . . . 103
3.17 Feasible scheduling on ST231, for reciprocal implementation. . . . 105

4.1 Special values for x/y. . . . 115

4.2 Performances of (non)restoring iterations with [without] subnormal numbers in RoundTiesToEven, on the ST231. . . . 125
4.3 Degree of polynomial approximant $a(t)$, and certified error bounds θ and η, for our *binary32* division implementation. . . . 127
4.4 Numerical estimation of $\alpha(a^*)$, for $6 \leq \delta \leq 11$. . . . 130
4.5 Coefficients of the polynomial approximant used for *binary32* division. . . . 130
4.6 Splitting steps. . . . 135

5.1 Coefficients of the polynomial $a(t)$ used for our square root implementation. . . . 148
5.2 Degree-7 polynomial evaluation using Estrin's rule on ST231. . . . 155
5.3 Evaluation error bounds for various functions and various fixed-point evaluation schemes. . . . 159
5.4 Signs of the coefficients of polynomial approximant $a(t)$, when $|n| = 2$. . . . 160
5.5 New coefficients for evaluating $P(s,t)$ using Paterson and Stockmeyer's algorithm, for our *binary32* square root implementation. . . . 164

6.1 Number of generated parenthesizations for evaluating a bivariate polynomial. . . . 177
6.2 Number of generated parenthesizations for evaluating particular bivariate polynomials. . . . 178
6.3 Number of parenthesizations of minimal latency (between brackets) for evaluating a bivariate polynomials. . . . 178
6.4 Number of parenthesizations of minimal latency (between brackets) for evaluating some special bivariate polynomials. . . . 179
6.5 Latency on unbounded parallelism and on ST231, for various functions of FLIP. . . . 183
6.6 Target latency for various degrees n_x and various delays D_s. . . . 184
6.7 First timings for the generation of parenthesizations for particular bivariate polynomials using our recursive search of best splittings. . . . 188
6.8 Impact of the value of the "keep" parameter on the timing of the generation and the "quality" ([latency, number of multiplications]) of the generated parenthesizations. . . . 188
6.9 Timing of the generation and quality ([latency, number of multiplications]) of the generated parenthesizations, for evaluating a degree-10 particular bivariate polynomial, $(n_x, n_y) = (9, 1)$. . . . 189
6.10 Timings for certified code generation for roots and their reciprocals, using this general framework. . . . 192
6.11 Timings for the generation of $\log_2(1+t)$ and $1/\sqrt{1+t^2}$. . . . 193
6.12 Performances on ST231 processor, of faithful implementations. . . . 196

List of algorithms

4.1 Restoring division for computing $\ell_0.\ell_1\ell_2\ldots\ell_{p-1}\ell_p$. 118
4.2 Nonrestoring division for computing $\ell_0.\ell_1\ell_2\ldots\ell_{p-1}\ell_p$. 120
4.3 Nonrestoring iteration using the `divs` and `addcg` instructions. 122

6.1 Bivariate(n_x, n_y). 170
6.2 BuildingRuleR1($k, \mathrm{P}^{(1)}, \ldots, \mathrm{P}^{(k-1)}$). 171
6.3 BuildingRuleR2($k, \mathrm{P}^{(1)}, \ldots, \mathrm{P}^{(k)}, \mathrm{E}^{(0)}, \ldots, \mathrm{E}^{(k-1)}$). 172
6.4 BuildingRuleR3($k, \mathrm{P}^{(1)}, \ldots, \mathrm{P}^{(k)}, \mathrm{E}^{(0)}, \ldots, \mathrm{E}^{(k)}$). 173
6.5 AB(n_x, n_y), minimal latency for evaluating $a'x^{n_x}y^{n_y}$. 183

Introduction

Today, embedded systems are everywhere, and are used in various application domains (multimedia, audio and video, or telecommunications, for example). Unlike general-purpose computers, embedded systems have microprocessors that are dedicated to one or a few specific tasks. Hence, they can be designed and tuned to satisfy several constraints: in terms of area and of energy consumption. And satisfying these constraints enables also to gain in terms of conception cost for designers. Currently, some embedded systems integrate their own dedicated floating-point units (FPU), but it is done to the detriment of an increase of its area and its energy consumption. Therefore to be able to satisfy such constraints, some embedded systems do not have any FPU, but only arithmetic and logical units (ALU's). However, that does not affect the performances of these architectures, which remain extremely efficient. Consequently, their low construction cost and energy consumption together with their small area and their efficiency, make embedded systems currently widely used, especially in real-time applications (audio and video, for example). But these applications are highly demanding on floating-point calculations, and since on this kind of architectures floating-point arithmetic is not implemented in hardware, it has to be emulated through a software implementation.

The problematics underlying this thesis is the design and the implementation of an efficient software support for IEEE 754 floating-point arithmetic on integer processors, through a set of mathematical operators offering correct rounding, handling of subnormal floating-point numbers, and handling of special inputs [IEE08]. Mostly, these implementations are particularly optimized for a specific target. The problem is that, by doing this "by hand", the developing time for such mathematical operators may be long (about up to 3 months for one function [Lau08, p. 197]), tedious, and also particularly error-prone. Furthermore each time a new target is developed or a new format is required, we have to rewrite some new software support for this format, optimized for that particular target. That statement has motivated the implementation of methodologies and tools dedicated to the automation of the implementation of floating-point operators, in hardware (FloPoCo [dDP]) and software (Sollya [CL], Metalibm [Lau], or CGPE [Rev]). And currently, the most important challenge aims at providing tools for generating in a fast way efficient and certified programs, optimized for the target on which they will be used.

This problematics has thus guided the work of this thesis, which aims first at providing some *efficient* and *certified* software support for *binary32* floating-point arithmetic (formerly called *single precision*) on integer processors, in particular VLIW's and DSP's, through a set of correctly-rounded basic algebraic functions, but also at providing some methodologies and tools that are expected to help the automation of the writing of some parts of these codes. Parts of this work has been done in collaboration with the *Compilation Expertise Centre* of *STMicroelectronics* (Grenoble, France), and validated on the processors of the ST200 family, which are 4-issue VLIW 32-bit embedded integer processors, more particularly the ST231 processor core. These processors are highly used in the audio and video domains, such as in set-top boxes for HD-IPTV (High Definition Internet Protocol Television), cell phones, wireless terminals, and PDAs.

To achieve these objectives, we bring out in this thesis several *basic blocks* from mathematical operator implementations. Hence the approaches that we propose here rely on a systematic use of these basic blocks, and thus enable to write quickly some efficient C codes. For each basic block, we give an algorithm parametrized by the target binary floating-point format as well as an analysis of the algorithm (especially those enabling computing correct rounding). Each of these basic blocks used is then illustrated with a C code for the *binary32* floating-point format and optimized for the ST231 processor core.

This work yield two software developments:

- A new, fully revised version of FLIP[1] (Floating-point Library for Integer Processors) that provides some portable C software support for *binary32* floating-point arithmetic on integer processors, particularly targeted to VLIW processors like the ST231.

- CGPE[2] (Code Generation for Polynomial Evaluation), which enables to write efficient and certified C code for the evaluation of bivariate polynomials in fixed-point arithmetic. The codes automatically produced by CGPE are integrated into those of FLIP.

We denote by "certified C code" a C code for which we can bound the evaluation error on integer arithmetic. To write such certified C codes, we proceed in two steps. First we compute with Sollya a polynomial approximant of the function to be implemented, as well as a certified approximation error and a certified evaluation error bound. Then we check with Gappa or by using MPFI that the evaluation error of our C code evaluating this polynomial satisfies the certified evaluation error bound. This process will be detailed later in the introduction.

The document is organized into two parts. Part I deals with the implementation in software of some binary floating-point operators optimized for the *binary32* format and for the ST231 processor. Particularly, we present a general approach for some of these operators based on the evaluation of a particular bivariate polynomial. In this first part, we will not discuss the way used for computing the polynomial evaluation programs, which is the subject of the second part. Indeed, in Part II, after a reminder of classical methods for evaluating polynomials and their applications in fixed-point arithmetic, we will present a methodology for generating efficient and certified C code for evaluating a given polynomial in fixed-point arithmetic. Now before detailing each of these parts, let us first give an outline of Chapter 1.

Chapter 1 - Context. This first chapter is an introductory chapter to the context of this work. Indeed, it aims at presenting some features (conditional branch reduction, called *if-conversion*) and constraints (instructions bundling, data accesses in memory) of the ST231 processor, both useful for analyzing and implementing functions (Part I), and optimizing and generating codes (Parts I and II). Then, it reminds the IEEE 754 standard for binary floating-point arithmetic, some basics about fixed-point arithmetic, and the way used to manipulate floating-point data on integer architectures.

Part I – Design of optimized algorithms for some binary floating-point operators and their software implementation on embedded VLIW integer processors

This part is focused on the optimized software implementation of some binary floating-point operators, targeted to embedded VLIW processors. So far, several software implementations of floating-point arithmetic have already been proposed, based on integer arithmetic:

[1] FLIP 1.0 is now available upon request.
[2] CGPE has not been released yet, but it is available upon request.

SoftFloat. In [Hau], SoftFloat provides a software implementation of the IEEE 754 binary floating-point arithmetic (single, double, double extended, and quadruple precision), for the four required rounding modes, with handling of exception flags and special values. For example, for the *binary32* floating-point format, in this library, square root is implemented using a method that refines a first approximation extracted from an array, while division relies on 64-bit integer division.

Software floating-point support of GCC. GCC provides some software floating-point support [GCC] containing implementations of the main arithmetic operators (addition, subtraction, multiplication, division, and negation), and where division is implemented using an *optimized* iterative method [EL94], that produces one bit of the result per iteration.

Glibc and μClibc. The GNU libc[3] integrates an implementation of several mathematical functions (soft-fp) for several binary formats (single, double, double extended), that can be used on processors having no FPU [SM06]. This is based on FDlibm (Freely Distributable LIBM [FDL]) which provides only double precision implementation of floating-point mathematical functions. Our interest in this library comes from the fact that it provides a software implementation of the *binary32* square root relying on integer arithmetic, which is based on an iterative method.

Remark that μClibc (also called uClibc) is a "light" version of the Glibc, optimized for embedded systems.[4] This library provides a libm where, for example, the single precision square root is implemented by calling double precision version and then casting the result to a single precision value. In terms of accuracy, that may lead to the problem of double rounding and thus to implementations that are *not* correctly-rounded.

GoFast Floating-Point Library. GoFast[5] is a family of fast IEEE 754 floating-point libraries, particularly designed to be used on embedded systems. For instance, for the single precision, it provides the addition, subtraction, multiplication, division, square root, and other elementary functions.

Hence, these methods are ill-suited to be used on ST231: first they do not exploit at best the instruction-level parallelism of the ST231. Second, due to the ST231 memory architecture, designers have to avoid methods based on look-up tables (this point will be explained in Section 1.1.4, Chapter 1), since they lead to a large increase of latency.

Originally, STMicroelectronics was using *STlib* as a software support for *binary32* floating-point arithmetic in their compilation tool chain. This library based on SoftFloat (without exception handling and subnormal support[6]) provides, in particular, a correctly-rounded software implementation of the five basic operations for the *binary32* floating-point arithmetic in RoundTiesToEven (rounding-to-nearest). In STlib, division and square root are implemented via iterative methods (see [EL94] or [PB02] for square root, or [OF97b] for division).

In [Rai06], Raina proposed a first improvement for these operators, by using multiplicative methods, Newton-Raphson or Goldschmidt (see [Mar04] for square root, or [Mar00] for division), that refine a first approximation of the function (or its reciprocal). This work has lead to the development of the library FLIP 0.3 [BBdD+04]. Table 1 below gives the performances of STlib and FLIP 0.3, in numbers of cycles and integer operations, for RoundTiesToEven, while Table 2 below shows the speedup of FLIP 0.3 compared to STlib. We can observe a first gain of between 20.8 %

[3] Available at `http://www.gnu.org/software/libc/`.
[4] Available at `http://www.uclibc.org/`.
[5] Information about GoFast can be found at `http://www.smxrtos.com/ussw/gofast.htm`.
[6] In STlib, subnormal inputs are considered as zeros.

and 73.3 %. However, these methods do not exploit at best the instruction-level parallelism of the ST231 processor. On the IBM Power3™ [AGS99], the evaluation of square root and division is done via the evaluation of a power series approximation (univariate polynomial), which enables to exploit more instruction-level parallelism than the iterative and multiplicative methods.

The contribution of this part have been integrated into FLIP 1.0, the new version of the FLIP library, which provides optimized and certified software implementation of several mathematical operators (addition, subtraction, multiplication, division, reciprocal, (reciprocal) square root, (reciprocal) cube root, ...) for the *binary32* floating-point format, with subnormal number support and for the four IEEE rounding-direction attributes (chosen at compile-time) required by the new version of the IEEE 754 standard [IEE08]. The objective was to be as compliant to the IEEE 754-2008 standard as possible. Today, only the exception flags are not supported in FLIP.

In Table 2 (last column), we can observe that the implementations of FLIP 1.0 are between 31.5 % and 48.8 % faster than the ones of the previous version, FLIP 0.3. To achieve these performances, our approach mainly consists in approximating the functions by a particular bivariate polynomial, and then in evaluating it as efficiently as possible on the ST231. The parenthesization chosen for evaluating this particular bivariate polynomial is automatically generated using CGPE (Part II).

	FLIP last development version (FLIP 1.0)				FLIP 0.3	STlib
	RN	RU	RD	RZ	RN	RN
addition	26 [87]	27 [92]	27 [91]	23 [73]	38 [114]	48 [165]
subtraction	26 [84]	26 [88]	26 [88]	23 [73]	39 [115]	49 [166]
multiplication	21 [71]	21 [73]	21 [72]	18 [59]	27 [89]	31 [86]
division	34 [103]	34 [107]	35 [106]	34 [101]	47 [116]	177 [*]
square root	23 [61]	23 [62]	23 [64]	23 [64]	45 [65]	95 [259]
reciprocal	25 [62]	27 [77]	26 [77]	26 [71]	41 [72]	-
reciprocal square root	29 [68]	29 [69]	29 [69]	29 [70]	56 [96]	-
cube root	34 [73]	43 [145]	45 [144]	42 [138]	-	-
reciprocal cube root	40 [86]	40 [94]	40 [94]	40 [88]	-	-
reciprocal fourth root	42 [81]	42 [82]	42 [83]	42 [83]	-	-

Table 1: Performances on ST231 in # cycles [# integer instructions].

[*] Division implementation cannot be flattened because of the presence of a call to 64-bit unsigned integer division.

For validation purposes (in terms of correctness of the implementations), the univariate functions of FLIP have been compared exhaustively against MPFR [FHL+07] and the Glibc. Concerning bivariate functions, they have been tested with the TestFloat package [Hau], and for the particular case of division, with the "Extremal Rounding Tests Set" [MM] as well.

Outline of Part I

This first part is organized into three chapters as follows.

Chapter 2 - Basic blocks for implementing correctly-rounded floating-point operators. This chapter presents the basic blocks we have brought out from implementations of correctly-rounded floating-point operators. More particularly, for each of them, it gives some parametrized

	FLIP 0.3 vs STlib	FLIP 1.0 vs STlib	FLIP 1.0 vs FLIP 0.3
addition	20.8 %	45.8 %	31.5 %
subtraction	20.4 %	46.9 %	33.3 %
multiplication	12.9 %	32.2 %	22.2 %
division	73.3 %	80.7 %	27.6 %
square root	52.6 %	75.7 %	48.8 %
reciprocal	-	-	39.0 %
reciprocal square root	-	-	48.2 %

Table 2: Speed-up of FLIP 0.3 compared to STlib (FLIP 0.3 vs STlib), of FLIP 1.0 compared to STlib (FLIP 1.0 vs STlib), FLIP 1.0 compared to FLIP 0.3 (FLIP 1.0 vs FLIP 0.3), in RoundTiesToEven*.

* Here, the implementation of FLIP 1.0 supports subnormal numbers.

descriptions and analyses. Furthermore, all along this chapter, C codes are proposed for illustrating them for the *binary32* floating-point format. All these codes have been optimized for the ST231. Finally, to show the interest of this preliminary definition and design of basic blocks, a complete example of C code is given for *binary32* multiplication.

Chapter 3 - A uniform approach for correctly-rounded roots and their reciprocals. This chapter presents a new approach for implementing roots and their reciprocals, which is based on the evaluation of a *single bivariate polynomial*. It turns out that this approach is more efficient on VLIW 4-issue architectures, like the ST231, since they exploit more instruction-level parallelism than classical ones. This efficiency is achieved thanks to the basic blocks of Chapter 2, an optimized polynomial evaluation code generated using the tools that will be described in Part II, the handling of special inputs in a way that fully exploits the binary interchange format encoding [IEE08, §3.4], and detailed and proved rounding procedures. Some elements of automation are given for constructing accurate-enough polynomial approximants with Sollya and for certifying their evaluation with Gappa. Finally, this approach is illustrated through the examples of *binary32* square root and reciprocal.

This work has lead to one publication at Asilomar'09 [JR09a] and to one submission to IEEE Transactions on Computers [JKMR08].

Chapter 4 - Extension to correctly-rounded division. This chapter extends the approach of Chapter 3 to division, and provides parametrized description and analysis of the division algorithm based on the evaluation of a *single bivariate polynomial*. Here again the method is presented with examples of C code for the *binary32* floating-point format and optimized for the ST231. Unlike roots and their reciprocals, the difficulties rely on the handling of special inputs (since division is bivariate), the validation of the polynomial evaluation program (which is less immediate), and the rounding condition implementation. These performances are achieved thanks to the same reasons as for roots and their reciprocals. Moreover, this chapter still points out the efficiency of this approach on ST231, even if a specific `divs` instruction is used (that computes in 1 cycle a nonrestoring iteration of division).

Parts of this work have been published at ARITH'19 [JKM+09].

Part II – Code generation for the efficient and certified evaluation of polynomials in fixed-point arithmetic

The approach we have proposed in Part I is mainly based on the evaluation of a particular bivariate polynomial that approximates accurately enough the function to be evaluated. The evaluation of this polynomial represents the main part of this approach, and thus dominates the global cost of the implementation. Hence, it may be critical for the efficiency of the code, and thus has to be optimized as much as possible.

Clearly, the number of evaluation schemes (parenthesizations) is extremely large, even for small polynomial degrees (≤ 5). Hence, choosing such parenthesizations "by hand" is long and tedious, and it is difficult to conclude on the optimality of the parenthesization kept for implementing the function. Therefore, we have proposed a tool called CGPE (for Code Generation for Polynomial Evaluation) that generates efficient and certified polynomial evaluation programs in fixed-point arithmetic. By efficient, we mean that the generated programs reduce the latency of evaluation on the considered target architecture, while by certified, we mean that the error entailed by the evaluation of the C code is no larger than a certified bound, so that we can ensure the correct rounding of all the operators implemented.

In [HKST99], a methodology is proposed for building optimal evaluation schemes for the evaluation of univariate polynomials on the Itanium® processor using only `fma` operations. Moreover, in [Gre02] a *brute force* method enables to product polynomial evaluation parenthesizations using at best SIMD instructions of the processor of PlayStation® 2 (Sony).

However, either these approaches are dependent to the target architecture, or use "naive" (brute force) algorithms to converge toward evaluation parenthesizations. Therefore this part discusses the efficient polynomial evaluation on integer architectures, especially on the ST231, and the generation of such efficient programs tuned according to the target architecture.

Outline of Part II

This second part is organized into two chapters, as follows.

Chapter 5 - Polynomial evaluation in fixed-point arithmetic on the ST231 processor. This chapter reminds some classical methods for evaluating polynomials (Horner's rule, "second-order" Horner's, Estrin's methods), and explains why they can be used efficiently for evaluating polynomials on the ST231. More particularly, it shows how to implement these evaluation schemes using only unsigned arithmetic (that may lead to a decrease of the evaluation latency and an accuracy that is twice better), and how to validate such evaluation programs with Gappa. As a conclusion, we observe that almost all these evaluation programs could have been used for implementing mathematical operators in the FLIP library. Finally, we explain why methods based on coefficient adaptations are not well-suited for evaluating polynomials on integer architectures. That statement is illustrated through two examples, the Knuth and Eve's algorithm and the Paterson and Stockmeyer's algorithm.

Chapter 6 - Computing efficient polynomial evaluation programs. This last chapter presents the methodology that we have designed for generating efficient polynomial evaluation programs. The interest of our approach relies on the fact that it is able to take (a simplified model of) the target architecture as an input parameter. Furthermore, it is based on an algorithm that computes all the parenthesizations for evaluating given polynomials, the determination of a lower bound of the minimal evaluation latency, and a heuristic that enables to search in a fast way for a set of *best* evaluation parenthesizations. This methodology has been integrated into CGPE, and has been validated for the implementation of several functions of

FLIP. For some of them, using this methodology, we can conclude that the polynomial evaluation program generated is optimal in terms of evaluation latency. Finally, the experimental framework presented here extends the works done in Chapters 3 and 4 on the automation of polynomial approximant computations and on the validation of efficient polynomial evaluation programs for general operators. Figure 1 illustrates this framework, which will be recalled and detailed in Chapter 6.

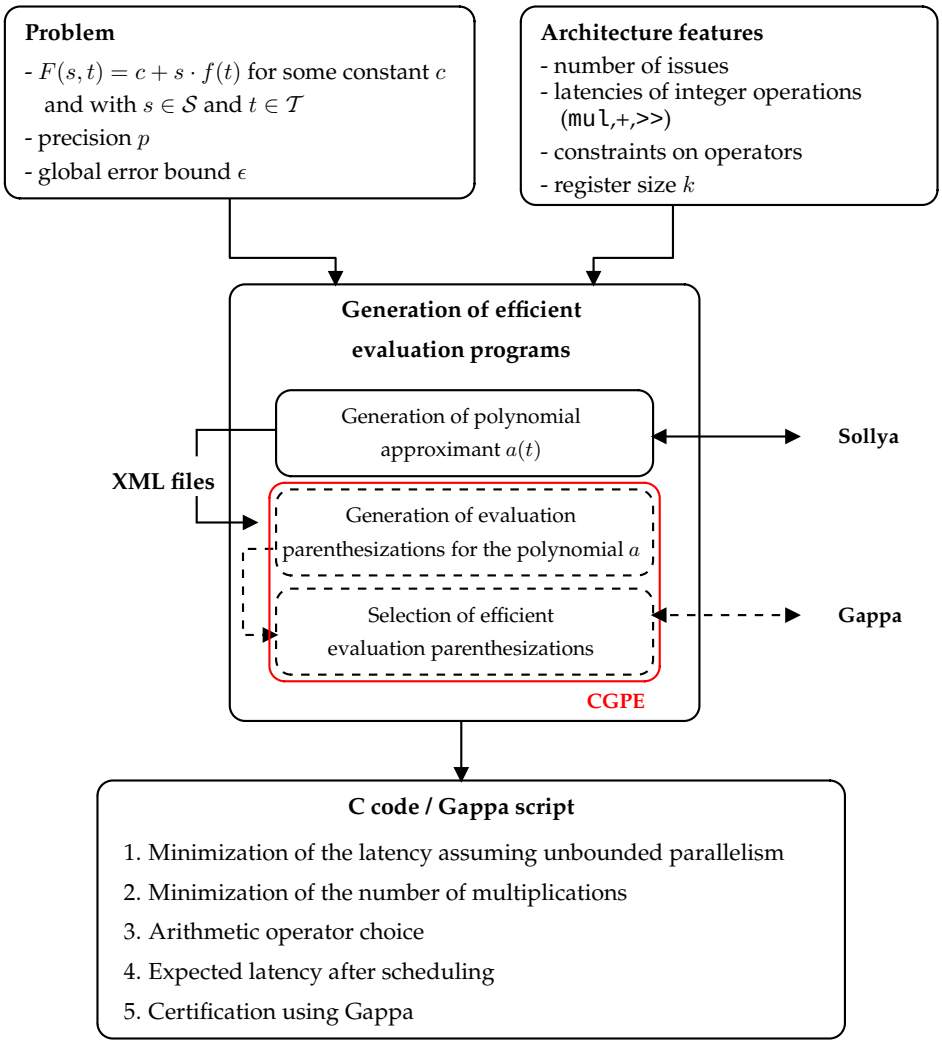


Figure 1: General framework for the automatic generation of efficient and certified polynomial evaluation programs.

Notice that throughout this document, the certification of our C code relies on the following tools:

- Sollya [CL] is a software environment created by Sylvain Chevillard [Che09] and Christoph Lauter [Lau08]. It is addressed to "anyone who needs to perform numerical computations in a safe environment." Particularly, the main features of Sollya we will use in this document are: the Remez algorithm, which approximates the "minimax" of a function on an interval ("minimax" will be defined in Definition 3.3, Chapter 3), and the certified supremum norm, which computes an interval enclosing the supremum norm of a function on a given interval (that will be used for computing the approximation error of a polynomial approximant with

respect to a given function). Examples of how to use these features are given in Chapters 3 (Listing 3.1) and 4 (Listing 4.6).

- Gappa [Mel] is a software tool developed by Guillaume Melquiond [Mel06]. It intends to help verifying and formally proving properties on numerical programs. Actually, it manipulates logical formulas involving the inclusion of expressions in intervals, and allows to bound rounding errors in a certified way. Gappa uses interval arithmetic and a set of theorems and rewriting rules for proving these properties, and providing a bound as tight as possible. We will denote by "Gappa certificate" a Gappa script corresponding to a given C code, so that the error bound computed by this script satisfies the required bound on the evaluation error of the considered C code. Examples of how to write Gappa certificates are given in Chapter 5 (Listing 5.3).

Chapter 1

Context

This chapter details the context of the work presented in this document. More particularly, it gives some key features (conditional branch reduction, called if-conversion*) and constraints (instructions bundling and data accesses in memory) of the target architecture, the ST231 processor, which is a 4-issue VLIW 32-bit integer processor. These elements will guide the techniques presented in Part I for providing efficient implementations of mathematical operators. This chapter further presents IEEE 754 binary floating-point arithmetic as well as fixed-point arithmetic.*

We have seen in introduction that the objective of our work is the design and the implementation of efficient software support for *binary32* floating-point arithmetic for integer processors. More precisely, we aim at providing a set of correctly-rounded mathematical functions, particularly optimized for the ST231 processor core, a specific 4-issue VLIW integer processor of the ST200 family. By definition, integer processors do not have floating-point units (FPU), and floating-point arithmetic thus has to be emulated, that is, implemented in software. Hence, given a mathematical function implementation, the input and output encode floating-point data, and internal computations are done using only operations on integers.

This first chapter is organized as follows. Section 1.1 gives an overview of the main features of the ST231 processor core. Then Section 1.2 presents IEEE 754 binary floating-point arithmetic as well as fixed-point arithmetic, and explains how to manipulate floating-point and fixed-point numbers on integer processors like the ST231.

1.1 Overview of the ST231 architecture

This first section presents the ST231 architecture, through a description of the main features of this target. After having given some key features, it explains the *if-conversion* mechanism, that allows the reduction of conditional branches, shows how the instructions are encoded, and then details the data accesses in memory. Finally, it simply gives some useful elements of the instruction architecture set.

1.1.1 Some features of the ST231 architecture and compiler

ST231, a 4-issue VLIW 32-bit processor

The ST231 is a 4-issue VLIW 32-bit integer processor and a member of the ST200 family core of STMicroelectronics. The VLIW microprocessors of ST200 family are embedded media processors,

that originate from the joint design of the Lx technology plateform by HP Labs and STMicroelectronics [FBF+00]. They are mainly designed to implement advanced audio and video codecs in consumer devices such as set-top boxes for HD-IPTV (High-Definition Internet Protocol Television), cell phones, wireless terminals, and PDAs. In particular, the ST231 is the most recently designed core of the ST200 family, and is widely used in STMicroelectronics SOCs for multimedia acceleration.

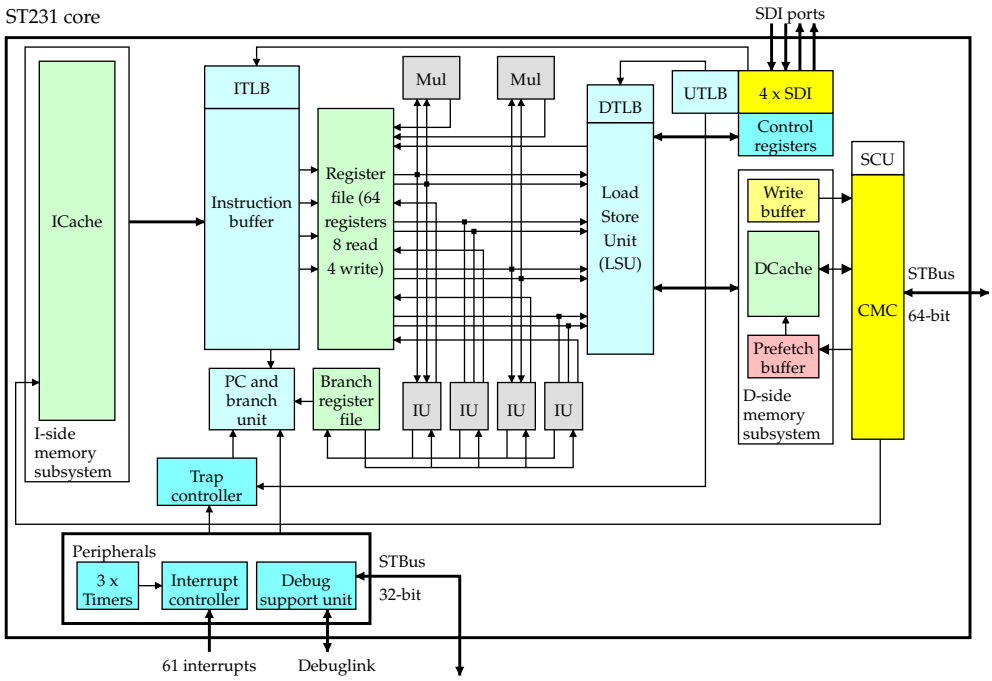


Figure 1.1: Block diagram of the ST231 architecture.

Figure 1.1 displays the block diagram of the ST231 architecture. As shown on this block diagram, the ST231 architecture includes the following features:

- parallel execution units, including four integer ALU's, Arithmetic Logic Units (or IU on Figure 1.1), and two pipelined 32×32 multipliers (Mul on Figure 1.1);
- a large register file of 64 general purpose 32-bit registers;
- efficient branch architecture with 8 one-bit *branch* registers (condition registers).

Concerning other features useful for efficiently implementing floating-point arithmetic, let us observe that the ST200 family includes:

- predicate execution through *select* operations;
- encoding of immediate operands up to 32 bits.

Architectural features of a VLIW processor

VLIW (Very Long Instruction Word) processors use an architectural technique where ILP (instruction-level parallelism) is explicitly exposed to the compiler [FFY05]. A VLIW processor issues simultaneously a fixed number of instructions, which were originally encoded in one large instruction word. On modern VLIW architectures, in particular on the ST231, the instructions are grouped

into a packet. In the ST231 terminology, this packet is called *bundle* and is composed of up to 4 instructions (see Section 1.1.3 for more details on the encoding of instructions). The processor can thus load the *bundle*, the packet of instructions, and issue them simultaneously. On the ST231, the RISC[1]-like operations in a same bundle are issued simultaneously and, since the delay between issue and completion is the same for all operations, complete simultaneously as well. This is the compiler that is in charge of bundling compatible instructions with each others, and thus of scheduling the code for the processor. (See [HP06, §2.7] or [Tan05, §8.1.1] for more details on VLIW architectures.)

A hardware implementation of a VLIW processor is significantly simpler than a corresponding multiple issue superscalar CPU. This is mainly due to the simplification of the operation grouping and scheduling hardware. This complexity is moved to the instruction-level parallelism extractor (compiler) and the instruction scheduling system (compiler and assembler) in the software toolchain.

Some key features of the ST231 compiler

The ST231 VLIW compiler is a C/C++ compiler based on the Open64 technology[2] retargeted for the ST200 processor family by STMicroelectronics. The compiler has been improved to support the development for high performance embedded targets. In our context, the two most important features of the ST231 compiler are the *if-conversion* and the *Linear Assembly Optimizer*, briefly described below.

If-conversion The *if-conversion* optimization enables to generate mostly straight-line assembly code by emitting efficient sequences of select instructions (`slct`) instead of costly control flow (this is detailed in Section 1.1.2);

Linear Assembly Optimizer In the context of *loop programming*, the *Linear Assembly Optimizer* (LAO) [dD04] generates a schedule of the instructions that is very close to the optimal. In our context, this generated schedule remains extremely efficient, and may be very close to the optimal, not to say optimal for polynomial evaluation.

(See [ST208a] for more details on ST231 VLIW compiler.)

1.1.2 *If-conversion*: reduction of conditional branches

To enable the reduction of conditional branches, the architecture provides partial predication support in the form of conditional selection instruction. This optimization is called *if-conversion*. To understand the interest of such an optimization, let us consider the following piece of C code (Listing 1.1 below), that returns -1 if $x < 0$, and 1 otherwise.

```
1 int32_t expl_ifconversion(int32_t x)
2 {
3   if( x < 0 ) return -1;
4   else        return  1;
5 }
```

Listing 1.1: Example of *if-conversion*.

The generated assembly code, using the ST231 C compiler (called `st200cc`), is presented in Listing 1.2 below. Indeed, we observe on Listing 1.2 that an instruction `slct` is used instead of a

[1] RISC: Reduced Instruction Set Computer.
[2] See `www.open64.net` for details.

```
1  cmplt $b0 = $r16, $r0          ## (cycle 0)
2  mov $r16 = -1                  ## (cycle 0)
3   ;; ## (bundle 0)
4  slct $r16 = $b0, $r16, 1       ## (cycle 1)
5  return $r63                    ## (cycle 1)
6   ;; ## (bundle 1)
```

Listing 1.2: Assembly code generated for the example of *if-conversion* (Listing 1.1).

conditional branch. In line 4, the instruction `slct` writes `$r16` in `$r16` if the branch register `$b0` is true ($x < 0$), and it writes `1` in `$r16` otherwise ($x \geq 0$).

This optimization of the generated assembly code is possible thanks to an efficient *if-conversion* algorithm based on the ψ-SSA representation, and used in the Open64 compiler to generate partially-predicated code based on the `slct` instruction [Bru06].

1.1.3 Bundle and instruction encoding

So far as we can, in our C codes, we will prefer to use "small" constants (9 signed bits), which may lead to a reduction of the evaluation latency as well as of the size of the generated assembly code. This section aims at presenting the way used for encoding instructions on the ST231 processor, and at explaining the advantages of using such small constants.

Description of instruction bundling and encodings

Instructions are encoded into *bundles* (wide words) containing from 1 to 4 instructions. More precisely, a *bundle* contains from 1 to 4 consecutive 32-bit words, called *syllables*. A syllable contains either an instruction or an *extended immediate* (for example, a 32-bit constant). A bundle is well-formed if it satisfies at least the following constraints:

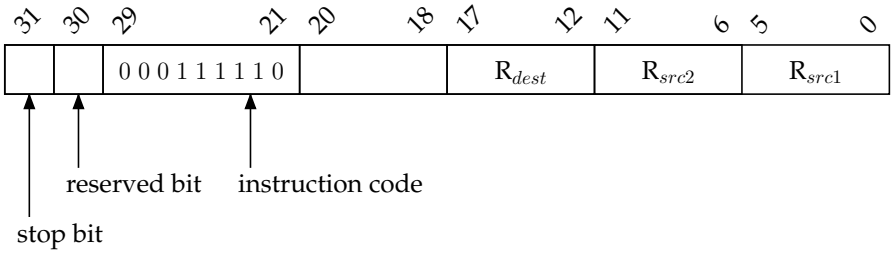
- all syllables of a bundle must be instructions or extended immediates;
- a bundle contains at most one control instruction, that must be the first syllable;
- a bundle contains at most one memory instruction (load/store);
- a bundle contains at most 2 multiplication instructions of the form 32-bit $\times$ 32-bit $\rightarrow$ 32-bit, that must appear at odd word addresses;
- in a bundle, immediate extensions must appear at even word addresses;
- in a bundle, no more than one immediate extension can be associated to a single instruction.

(See [ST208b, §6.6.1] for more details on bundling constraints.)

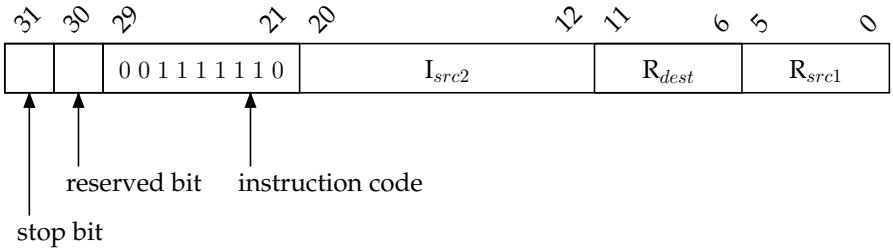
Many instructions have an *immediate* form. A specificity of the ST231 processor is that these immediate forms are by default encoded to use small immediates (9-bits signed), but can be extended to use extended immediates (32-bits) at the cost of one syllable per immediate extension. This makes the usage of extended immediate constants (such as 32-bit polynomial coefficients) very efficient from a memory system standpoint. In this case, the immediate extension is encoded in a word adjacent to the one encoding the instruction, either on the left or on the right, in the bundle.

Example of instruction encoding

Let us consider for example the instruction `mul64hu`, that returns the 32 most-significant bits of the product of two 32-bit unsigned integers. On the ST231 processor, this instruction can be used either in *register* form or in *immediate* form. Figure 1.2 below shows for both forms how this instruction is encoded into a syllable.



(a) *Register* form.



(b) *Immediate* form.

Figure 1.2: The two possible encodings of the `mul64hu` instruction.

In *register* form (Figure 1.2(a)), the addresses of input and output registers are encoded with 6 bits each (which is sufficient to encode the addresses of the 64 available registers), between the bits of weigth 17 and 0 of the syllable. In *immediate* form (Figure 1.2(b)), the addresses of the output and one of the input registers are encoded in the same way (6 bits each), between the bits of weigth 11 and 0 of the syllable, while the immediate constant is encoded with 9 bits (between the bits of weigth 20 and 12).

In C language, the instruction `mul64hu` can be called using the function `mul` of Appendix B (see [ST208b, pp. 248–249] for specifications). Given X and Y two k-bit unsigned integers (here, $k = 32$), we have:

$$\texttt{mul}(X, Y) = \lfloor (X \cdot Y)/2^k \rfloor, \tag{1.1}$$

where $\lfloor \cdot \rfloor$ denotes the usual floor function. Let us now consider the following piece of C code, that uses the function `mul`.

```
1 uint32_t expl1(uint32_t x, uint32_t y)
2 {
3   uint32_t r1 = mul(x, 0xb540304f);
4   uint32_t r2 = mul(x, 0x804f);
5   uint32_t r3 = x + y;
6   return ((r1 + r2) + r3);
7 }
```

Listing 1.3: First example of `mul64hu` use.

In this first example, the two multiplications have a second operand that *cannot* be encoded in small immediate constants (9-bit signed). Thus, the instruction `mul64hu` in the generated assembly code is in the *register* form, and both operands are encoded in separate syllables. It follows that

```
1  mul64hu $r18 = $r16, 32847              ## (cycle 0)
2  mul64hu $r19 = $r16, -1254084529        ## (cycle 0)
3   ;; ## (bundle 0)
4  add $r16 = $r16, $r17                   ## (cycle 1)
5   ;; ## (bundle 1)
6  add $r17 = $r18, $r19                   ## (cycle 3)
7   ;; ## (bundle 2)
8  add $r16 = $r16, $r17                   ## (cycle 4)
9  return $r63                             ## (cycle 4)
10  ;; ## (bundle 3)
```

Listing 1.4: Assembly code generated for the first example of `mul64hu` use (Listing 1.3).

the instructions in lines 3 and 4 of Listing 1.3 above fill up the first bundle and the addition at line 5 of Listing 1.3 has to be launched in the second bundle, as can be seen on the piece of assembly code in line 4 of Listing 1.4.

Let us now consider a second example. In this second example, we replace line 4 of Listing 1.3 by the following line of C code:

```
uint32_t r2 = mul(x, 0x4f);
```

Here, the second operand of the multiplication is between -256 and 255 (indeed, $0x4f_{16} = 79$), and it can be encoded directly in the instruction's syllable as a small immediate constant. Hence this multiplication operation is in *immediate* form in the generated assembly code (Listing 1.5 below).

```
1  mul64hu $r18 = $r16, 79                 ## (cycle 0)
2  add $r8 = $r16, $r17                    ## (cycle 0)
3  mul64hu $r16 = $r16, -1254084529        ## (cycle 0)
4   ;; ## (bundle 0)
5  add $r16 = $r18, $r16                   ## (cycle 3)
6   ;; ## (bundle 1)
7  add $r16 = $r8, $r16                    ## (cycle 4)
8  return $r63                             ## (cycle 4)
9   ;; ## (bundle 2)
```

Listing 1.5: Assembly code generated for the second example of `mul64hu` use.

It follows now that the two multiplications fill up only three of the four syllables of the first bundle: two for the multiplication instructions, and one for the extended immediate ($0xb540304f_{16}$), while the small immediate constant is encoded direcly in the instruction's syllable. Therefore the addition in line 5 (Listing 1.3 above), which does not have constant operands, can be also added into the first bundle, and is issued simultaneously with the two multiplication operations (see Listing 1.5 above).

Concerning code size, in Listing 1.3, we have 6 operations (2 multiplications, 3 additions, and 1 return) and 2 immediate extensions, encoded on 32 bits each: the code size is 8×32 bits = 32 bytes. On the contrary, on the second example (after having replaced the multiplication of line 4 in Listing 1.3), we have also 6 operations, but only 1 immediate extension: the code size is only of 7×32 bits = 28 bytes. We gain 4 bytes (that is, 32 bits) by using a small immediate.

Obviously, on the piece of assembly code in Listing 1.5 above, the operand registers `$r16` in lines 1, 2, and 3 correspond to the same value and are used in cycle 0. However, the result register `$r16` in line 3 will be effectively written and available at cycle 1. Hence, there is no conflict between writing in `$r16` in line 3 and reading this same register in lines 1, 2, and 3.

Remark that in ST200 assembly code, the `return` instruction at the end of a procedure (line 8 of Listing 1.5 above, for example) corresponds to the **`return`** operation of the corresponding C code. More particularly, the instruction `return $r63` in Listing 1.4 or Listing 1.5 is a branch into the calling frame of the assembly code (or the program), just after the call point of the function. The address corresponding to that particular point is stored at call time into the register `$r63`, also called *Link Register* (see [ST208b] for details and restrictions on Link Register). The latency of such a `return` instruction is the same as any classical branch (that is, 1 cycle). However, the Link Register has to be written at least 3 cycles before this "branch", otherwise the ST231 stalls until the address stored in `$r63` is known. Note that this constraint has no impact on code of at least 3 cycles, which is the case of the implementations we will present in this document.

How to code cleverly by using small immediate constants?

Until now we have observed the impact on the generated assembly code of using small immediate constants, and in particular on code size. Let us now have a look at how to write code cleverly by using small immediate constants. Consider the piece of C code in Listing 1.6 below. On this small

```
1 uint32_t expl3(uint32_t x, uint32_t y)
2 {
3   uint32_t r1 = mul(x, 0x00020000); // 0x00020000 = 2^17
4                                     // -> does not fit in a small immediate
5   uint32_t r2 = r1 + 0xb540304f;
6   return r2;
7 }
```

Listing 1.6: Example of C code without small immediate constant.

example, assuming that multiplication takes 3 cycles and that addition takes 1 cycle, the result `r2` is obtained after 4 cycles, as shown on the assembly code in Listing 1.7 below. However, since

```
1   mul64hu $r16 = $r16, 131072     ## (cycle 0) -> r1 = mul(x, 0x00020000)
2    ;; ## (bundle 0)
3   add $r16 = $r16, -1254084529    ## (cycle 3) -> r2 = r1 + 0xb540304f
4   return $r63                     ## (cycle 3)
5    ;; ## (bundle 1)
```

Listing 1.7: Assembly code generated for the third example of `mul64hu` use (Listing 1.6).

$0x00020000_{16} = 2^{17}$, by definition of the function `mul` we have

$$\texttt{mul}(x, 2^{17}) = \lfloor (x \cdot 2^{17})/2^{32} \rfloor = \lfloor x \cdot 2^{-15} \rfloor.$$

And in integer arithmetic $\lfloor x \cdot 2^{-15} \rfloor$ may be computed by shifting x right by 15 bits. Hence the line 3 of Listing 1.6 above may be replaced by the following line of C code.

```
uint32_t r1 = x >> 15;
```

And finally, assuming that the "shift right" operation has a latency of 1 cycle, the result `r2` is available after only 2 cycles, as shown in Listing 1.8 below. Remark also that using the shift operation (instead of multiplication) leads to a gain of 4 bytes, since its second operand (which in our example is equal to 15) can be encoded in a small immediate.

Note finally that in the small piece of assembly code in Listing 1.8 below, the instruction **`add`** in line 3 depends directly on the instruction `shru` in line 1, since the register `$r16` written in line 1 is used as operand of instruction **`add`** in line 3.

```
1  shru $r16 = $r16, 15            ## (cycle 0) -> r1 = x >> 15
2   ;; ## (bundle 0)
3  add $r16 = $r16, -1254084529    ## (cycle 1) -> r2 = r1 + 0xb540304f
4  return $r63                     ## (cycle 1)
5   ;; ## (bundle 1)
```

Listing 1.8: Assembly code generated for the example of the "shift right" operation.

1.1.4 Data accesses in memory

For the implementation of mathematical functions, some methods, called *look-up table methods*, rely on the refinement of a first approximation of the function (or its reciprocal) extracted from an array. However, the memory architecture is ill-suited for such methods. Let us now give some explanations.

All the datum accesses pass by the *Load Store Unit* (LSU), and are handled in the *D-side subsystem memory*. The *D-side subsystem memory* consists of a *writer buffer*, a *data cache*, and a *prefetch buffer* (see Figure 1.1 above). The *data cache* is 32-Kbyte 4-way associative cache. (See [Tan05, pp. 319-321] or [HP06, §5] for more details on associative cache.) More precisely, the *DCache* is split up into 256 sets of 4 lines (one per way) of 32 bytes each. And the prefetch buffer contains 8 entries, with 32 data bytes each.

When the *Load Store Unit* requests a datum access, the following three cases may occur.

Load from DCache. If the datum is in the data cache (DCache), it is loaded directly by the *Load Store Unit*. The latency of this load operation is of 3 cycles.

Load from uncached region of memory. If the datum is in an uncached region of memory (in external memory, for example), it is transferred into the DCache by the STBus, and then to the LSU. More precisely, the DCache stalls until the transfer from uncached memory has completed: this is called *data cache miss*. When the datum is in an uncached region of memory, the loading operation may take about 100 cycles.

Load from prefetch buffer. Finally, if the datum is in the *prefetch buffer*, it is transferred into the DCache. In fact, the prefetch buffer "prefetches and stores data from external memory and sends it to the data cache when (and if) it is necessary" [ST208b, §7.3.6]. In this case, the loading operation takes about 10 cycles, and the cost of the transfer of data from uncached memory to prefetch buffer is hidden, since it is not done at loading time (but before).

Therefore it follows that loading a datum from memory may be highly costly, especially if it is still in external memory. From Table 1, in Introduction, we see for example that our implementation of the square root operation (detailed in Chapter 3, Section 3.5) has a latency of 23 cycles: adding about 100 cycles would lead to an implementation about 5 times slower. Hence in C language, we will avoid to use arrays, and to read data from them. More precisely, when implementing a mathematical function on this kind of architecture, we will not use methods based on look-up tables (see [Mar90], for square root, for example), but prefer those based on small degree polynomial evaluation [Rai06] for the first approximation of multiplicative methods, or based on bivariate polynomial evaluation (detailed in Chapters 3 and 4), where evaluation points may be computed using logical operations (see Section 3.5.3 for example).

1.1.5 Elements of the instruction architecture set

Table 1.1 below summarizes the ST231 instructions that we mostly used when designing and implementing the algorithms in the next chapters, and explains how to use them in C language.

Instruction	Description / Result	Use in C language
`add`, `sub`	Addition, Subtraction	`X + Y`, `X - Y`
`addcg`	Addition with carry (ci) and generate carry (co)	`__ADDCG(R,co,X,Y,ci)` (see Section 4.4.3 for an example)
`and`	Bitwise AND	`X & Y`
`andl`	Logical AND	`X && Y`
`clz`	Count leading zeros	`nlz(X)` (see Section B.4)
`divs`	Nonrestoring division step	`__DIVS(Wj,qj,Wj,M,0)` (see Section 4.4.3 for an example)
`mul32`	Low part (32-bit) of a 32×32 signed product	`X * Y`
`mul64h`	High part (32-bit) of a 32×32 signed product	`mul64h(X,Y)` (see Section B.3)
`mul64hu`	High part (32-bit) of a 32×32 unsigned product	`mul(X,Y)` (see Section B.3)
`or`	Bitwise OR	`X | Y`
`orl`	Logical OR	`X || Y`
`max`, `maxu`	Signed, unsigned maximum	`max(X,Y)`, `maxu(X,Y)` (see Section B.1)
`min`, `minu`	Signed, unsigned minimum	`min(X,Y)`, `minu(X,Y)` (see Section B.2)
`shBadd`	Shift left B bits and accumulate, B $\in \{1,2,3,4\}$	`(X « B) + Y`
`shl`, `shr`	Shift left, shift right	`X « B`, `X » B`
`slct`, `slctf`	Select instructions	`if(cond){...}else{...}` (see Section 1.1.2)
`xor`	Bitwise XOR, exclusive-or	`X ^ Y`

Table 1.1: Elements of the ST231 instruction architecture set.

All these instructions have a latency of 1 cycle, except the multiplication instructions (`mul32`, `mul64h`, and `mul64hu`) which have a latency of 3 cycles. Recall that on the ST231 all multiplication instructions are fully pipelined.

1.2 Binary floating-point and fixed-point arithmetics

This section presents the binary arithmetics used all along this document to approximate the real numbers. More particularly, it presents first the *binary floating-point arithmetic* defined in the IEEE 754-2008 standard [IEE08]. Since the target architecture manipulates only integer numbers, we then present the *binary interchange format encoding* used to represent *floating-point data* with integers. Finally we give some elements on how to use the available integer arithmetic to implement *fixed-point arithmetic*.

1.2.1 Binary floating-point arithmetic

In 1985, a first standard for binary floating-point arithmetic was published [Ame85], called IEEE 754-1985. This first version defined four binary floating-point formats to represent real numbers: *single*, *single extended*, *double*, *double extended*. Since then, this standard has been revised, and a new version, called IEEE 754-2008, was published in August 29, 2008.

Binary floating-point formats and data

The IEEE 754-2008 standard [IEE08] characterizes a *floating-point format* by a radix β, a precision p, and an exponent range $\{e_{\min}, \ldots, e_{\max}\}$, such that

$$e_{\min} = 1 - e_{\max}. \tag{1.2}$$

Actually it defines two kinds of formats: the *binary* ($\beta = 2$) and the *decimal* ($\beta = 10$) floating-point formats. In all this document, we will consider only *binary floating-point formats*, that is, radix $\beta = 2$.

More particularly, the IEEE 754-2008 standard defines three basic *binary floating-point formats*: *binary32*, *binary64*, and *binary128*. Table 1.2 gives the parameters of these three basic formats. (In the context of the implementation of mathematical functions in the FLIP library, we will

	binary32	*binary64*	*binary128*
precision p of the format	24	53	113
extremal exponents $e_{\min}$, $e_{\max}$	-126, 127	-1022, 1023	-16382, 16383

Table 1.2: Basic binary floating-point formats parameters.

give all code examples in the *binary32* floating-point format.)

The floating-point formats allow to represent a finite subset of the set of extended real numbers as well as "non numbers". More particularly, each floating-point format represents a unique set of *floating-point data*. Let x be a *binary floating-point datum*. In a given floating-point format, the floating-point datum x can be:

- either a *special number*: signed zero (± 0), signed infinity ($\pm\infty$), or a "not-a-number" (NaN);
- or a *binary floating-point number*.

Recall that the standard does not interpret the sign of a NaN [IEE08, §6.3].

Binary floating-point numbers

A *nonzero binary floating-point number* x is defined by a sign s_x, an exponent e_x, and a significand m_x, as follows:

$$x = (-1)^{s_x} \cdot m_x \cdot 2^{e_x}, \tag{1.3}$$

with $s_x \in \{0, 1\}$, $e_x \in \{e_{\min}, \ldots, e_{\max}\}$, and $m_x = m_{x,0}.m_{x,1} \ldots m_{x,p-1}$, with $m_{x,i} \in \{0, 1\}$, $0 \leq i < p$. From now on, we denote by *normal binary floating-point number* (or *normal number*, for short), a floating-point number x defined such as:

$$m_x = 1.m_{x,1} \ldots m_{x,p-1}.$$

Note that $m_x \in [1, 2 - 2^{1-p}]$. Writting Ω for the largest normal floating-point number, it follows that every normal number satisfies:

$$2^{e_{\min}} \leq |x| \leq \Omega \quad \text{with} \quad \Omega = 2^{e_{\max}} \cdot (2 - 2^{1-p}).$$

Now, let x be a nonzero binary floating-point number with magnitude less than $2^{e_{\min}}$. Then, necessarily,

$$m_x = 0.m_{x,1} \ldots m_{x,p-1} \in [2^{1-p}, 1 - 2^{1-p}] \quad \text{and} \quad e_x = e_{\min}.$$

Such a floating-point number is called *subnormal binary floating-point number* (or *subnormal number*, for short). We observe that for a given format, a subnormal number x is as follows, with α the smallest positive subnormal floating-point number:

$$\alpha \leq |x| \leq 2^{e_{\min}}(1 - 2^{-p}) \quad \text{with} \quad \alpha = 2^{e_{\min}-p+1}.$$

Remark that subnormal floating-point numbers always have fewer than p significant bits.

Normalized representation of a binary floating-point number

When implementing a mathematical function, we do not want to distinguish between these two cases for efficiency reasons, and we prefer handling normal and subnormal floating-point numbers together. Hence, one of the first steps of this kind of implementation may consist in computing what we call the *normalized significand* of the floating-point input. To do so, let λ_x be the number of leading zeros in the binary expansion of m_x. If x is a normal number, then $\lambda_x = 0$. Otherwise, if x is a subnormal floating-point number, we have $\lambda_x \in \{1, \ldots, p-1\}$. We finally get:

$$m_x = \underbrace{0.00\ldots00}_{\lambda_x}\underbrace{1m_{x,\lambda_x+1}\ldots m_{x,p-1}}_{p-\lambda_x} \quad \text{with} \quad \lambda_x \in \{0, \ldots, p-1\}.$$

Let m'_x and e'_x be respectively the normalized significand and the scaled exponent of a binary floating-point number x:

$$m'_x = m_x \cdot 2^{\lambda_x} \quad \text{and} \quad e'_x = e_x - \lambda_x. \tag{1.4}$$

Therefore all along this document, we will call a *nonzero (sub)normal binary floating-point number*, a floating-point number as follows:

$$x = (-1)^{s_x} \cdot m'_x \cdot 2^{e'_x}, \tag{1.5}$$

with e'_x and m'_x as above. Notice that $e'_x \in \{e_{\min} - p + 1, \ldots, e_{\max}\}$ and that $m'_x \in [1, 2 - 2^{1-p}]$.

Finally, following [MBdD+09, §8], we will denote by n_x the "is normal bit" of x, defined as follows:

$$n_x = \begin{cases} 0, & \text{if } x \text{ is subnormal,} \\ 1, & \text{if } x \text{ is normal.} \end{cases} \tag{1.6}$$

1.2.2 Binary interchange format encoding

This section details the binary interchange format encoding defined in [IEE08, §3.4] to encode a binary floating-point datum into a k-bit unsigned integer.

Standard encoding of binary floating-point numbers

Using the *binary interchange format encoding* defined in the IEEE 754-2008 standard [IEE08, §3.4], a binary floating-point number is represented by three fields:

- a sign bit,
- a w-bit biased exponent $E_x = e_x - e_{\min} + n_x$,
- a $(p-1)$-bit trailing significand field T_x,

as shown in Figure 1.3. In [IEE08, §3.4], the exponent width w is defined as:

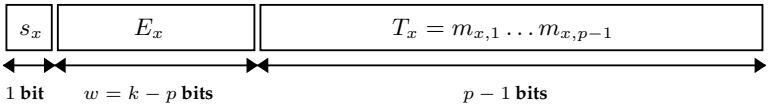


Figure 1.3: Binary interchange format encoding of floating-point data.

$$w = k - p. \tag{1.7}$$

By definition of subnormal numbers, if x is subnormal floating-point number, then its exponent e_x equals $e_{\min}$. Hence it follows that its biased exponent is $E_x = 0$. More generally, we have

$$E_x \in [0, 2^w - 1],$$

and E_x can be encoded with a w-bit unsigned integer.

Table 1.3 gives the encoding parameters for the three basic binary floating-point formats. Throughout this document, we will give some code examples for the *binary32* floating-point for-

	binary32	*binary64*	*binary128*
k, encoding bit-width	32	64	128
p, precision of the format	24	53	113
$e_{\max}$, maximal exponent	127	1023	16383
$e_{\min}$, minimal exponent	-126	-1022	-16382
w, biased exponent bit-width	8	11	15

Table 1.3: Interchange format parameters for the basic binary floating-point formats.

mat, that is, for $(k, p, e_{\max}) = (32, 24, 127)$.

Let X be the k-bit unsigned integer encoding the floating-point number x. The bit string of X is as follows: X_{k-1} encodes the sign of x, the next w bits $X_{k-2}, \dots, X_{p-1}$ encode the biased exponent E_x, and the last $p-1$ bits encode the trailing significand of x. We get finally:

$$X = S_x + E_x \cdot 2^{p-1} + (M_x - n_x \cdot 2^{p-1}),$$

with $S_x = s_x \cdot 2^{k-1}$ and $M_x = m_x \cdot 2^{p-1}$.

(Details are given in Section 2.1 about the encoding of a floating-point number into an unsigned integer, and more particularly on how to use this encoding efficiently.)

Example 1.1. *Consider the* binary32 *floating-point number*

$$\begin{aligned} x &= 5.656854152679443359375 \\ &= (-1)^0 \cdot 2^2 \cdot 1.41421353816986083984375. \end{aligned}$$

We have $s_x = 0$, $e_x = 2$, *and*

$$\begin{aligned} m_x &= 1.41421353816986083984375 \\ &= 1.01101010000010011110011_2, \end{aligned}$$

It follows that $n_x = 1$, $S_x = 0$, $E_x = 129 = 10000001_2$, *and* $M_x = 10110101000001001111001 1_2$. *Hence, the 32-bit unsigned integer* X *encoding the floating-point number* x *is as follows:*

$$\boxed{0 \;\; 10000001 \;\; 01101010000010011110011}.$$

Encoding of special data

Recall that binary floating-point data can also be either a signed zero (± 0), a signed infinity ($\pm\infty$), or a "not a number" (NaN). For each of these special data, the IEEE 754-2008 standard defines a specific binary interchange encoding, which is not always unique. Let X be a k-bit unsigned integer encoding of a special datum. Then,

- Signed zeros are encoded with integers of the form

$$X = S_x, \quad \text{with} \quad S_x = s_x \cdot 2^{k-1},$$

  and s_x the sign bit of x;

- Signed infinities are encoded with integers of the form

$$X = S_x + (2^{k-1} - 2^{p-1}), \quad \text{with} \quad S_x = s_x \cdot 2^{k-1},$$

  and s_x the sign bit of x;

- "Not-a-Numbers" are encoded with integers of the form

$$X = S_x + (2^{k-1} - 2^{p-1}) + T_x, \quad \text{with} \quad S_x = s_x \cdot 2^{k-1},$$

  and $s_x \in \{0, 1\}$, and $T_x \neq 0$.

The IEEE 754-2008 standard defines two kinds of "Not-a-Numbers": the signaling NaN's (sNaN) and the quiet NaN's (qNaN), that have both a specific encoding. In particular, let X be a k-bit unsigned integer encoding of a floating-point datum x, as above. Then x is a qNaN if and only if the bit of weigth $p-2$ of T_x equals 1. Otherwise, x is an sNaN.

Example 1.2. *Consider the* binary32 *floating-point format, that is,* $(k, p, e_{max}) = (32, 24, 127)$. *The*

following integers encode respectively -0, $+0$, $-\infty$, $+\infty$, an sNaN, and a qNaN.

-0	*1 00000000 00000000000000000000000*
$+0$	*0 00000000 00000000000000000000000*
$-\infty$	*1 11111111 00000000000000000000000*
$+\infty$	*0 11111111 00000000000000000000000*
sNaN	*1 11111111 01101010000010011110011*
qNaN	*0 11111111 11101010000010011110011*

The relationship between a floating-point datum and its encoding into integer is summarized in Table 1.4, that displays the relationship between a floating-point datum and its encoding into a k-bit unsigned integer X.

Remark that a qNaN should be used to propagate information out of an implementation, such as invalid or unavailable data and result. For example, in the implementation of a mathematical function, when an invalid operation occurs and the function has to return a floating-point result, the IEEE 754-2008 standard recommends that a qNaN be returned. To facilitate the *a posteriori* diagnostic, a qNaN should preserve as much information of the inputs as possible in particular when quieting a sNaN [IEE08, §6.2]. Here and hereafter we call *payload* the "diagnostic information contained in a NaN, encoded in part of its trailing significand field" [IEE08, p. 4]. Typically, the payload of a NaN is encoded into the last $p-2$ bits of its trailing significand field.

1.2.3 Binary fixed-point arithmetic

Until now we have recalled some features of IEEE 754 binary floating-point arithmetic, and we have seen how to use k-bit unsigned integers to store floating-point data. Moreover, since the ST231 architecture has no FPU, the implementation of binary floating-point operators relies on computations using available integer arithmetic on integers and *fixed-point numbers*.

The *binary fixed-point arithmetic* defined here consists in interpreting an integer as a rational number in a given format. In particular, that rational number is obtained by multiplying this integer by a given power of 2, which depends directly on the format. We recall here two ways to encode a given real value in an integer using fixed-point arithmetic [EL04, §1].

Unsigned value representation

The simplest case occurs when the value represented by the integer is non negative, thus we do not have to reserve one bit to handle the sign. To do so, let X be a k-bit unsigned integer encoding a positive value x. The value x is in the format Q_{f} if it is of the form:

$$x = X \cdot 2^{-f} \quad \text{with} \quad 0 \leq X \leq 2^k - 1.$$

Here f denotes the number of fraction bits in the binary expansion of x. If $i = k - f$ is the number of bits of the integer part of x, then X encodes the real value x in the $Q_{\mathrm{i.f}}$ format. The binary expansion of x is as follows:

$$x = X_{k-1} \cdots X_f . X_{f-1} X_{f-2} \cdots X_1 X_0.$$

Value or range of integer X	Floating-point datum x	Bit string $X_{k-1}\ldots X_0$		
		X_{k-1}	$X_{k-2}X_{k-3}\cdots X_{p-1}$	$X_{p-2}X_{p-3}\cdots X_0$
0	$+0$	0	$00\cdots00$	$0000\cdots0000$
$(0, 2^{p-1})$	positive subnormal number	0	$00\cdots00$	$X_{p-2}X_{p-3}\cdots X_1X_0$ with some $X_i = 1$
$[2^{p-1}, 2^{k-1} - 2^{p-1})$	positive normal number	0	$\underbrace{X_{k-2}X_{k-3}\cdots X_{p-1}}$ not all ones, not all zeros	$X_{p-2}X_{p-3}\cdots X_1X_0$
$2^{k-1} - 2^{p-1}$	$+\infty$	0	$11\cdots11$	$0000\cdots0000$
$(2^{k-1} - 2^{p-1}, 2^{k-1} - 2^{p-2})$	sNaN	0	$11\cdots11$	$0X_{p-3}\ldots X_0$ with some $X_i = 1$
$[2^{k-1} - 2^{p-2}, 2^{k-1})$	qNaN	0	$11\cdots11$	$1X_{p-3}\ldots X_0$
2^{k-1}	-0	1	$00\cdots00$	$0000\cdots0000$
$(2^{k-1}, 2^{k-1} + 2^{p-1})$	negative subnormal number	1	$00\cdots00$	$X_{p-2}X_{p-3}\cdots X_1X_0$ with some $X_i = 1$
$[2^{k-1} + 2^{p-1}, 2^k - 2^{p-1})$	negative normal number	1	$\underbrace{X_{k-2}X_{k-3}\cdots X_{p-1}}$ not all ones, not all zeros	$X_{p-2}X_{p-3}\cdots X_1X_0$
$2^k - 2^{p-1}$	$-\infty$	1	$11\cdots11$	$0000\cdots0000$
$(2^k - 2^{p-1}, 2^k - 2^{p-2})$	sNaN	1	$11\cdots11$	$0X_{p-3}\ldots X_0$ with some $X_i = 1$
$[2^k - 2^{p-2}, 2^k)$	qNaN	1	$11\cdots11$	$1X_{p-3}\ldots X_0$

Table 1.4: Relationship between floating-point datum x and its encoding into integer X ($k = p + w$).

It is well-known that the dynamic of the binary fixed-point numbers is lower than the one of binary floating-point numbers, and it is much more difficult to do computations between fixed-point numbers of different orders of magnitude. With this representation, we can represent the subset of real values defined as follows:

$$\left\{F \cdot 2^{-f}\right\}_{F=0,1,\ldots,2^k-1}.$$

Example 1.3. *Assume $k = 32$. Let X be the 32-bit unsigned integer that encodes the real value x, with*

$$X = 10110101000001001111001100110100_2.$$

If the integer X encodes the real value in the format $Q_{1.31}$, we deduce that:

$$\begin{aligned} x &= X \cdot 2^{-31} \\ &= 10110101000001001111001100110100_2 \cdot 2^{-31} \\ &= 1.01101010000010011110011001101002 \\ &= 1.414213562384247779846191406 25. \end{aligned}$$

However, if we consider that X encodes the real value x with 17 fraction bits, that is, in the format $Q_{15.17}$, we have:

$$\begin{aligned} x &= X \cdot 2^{-17} \\ &= 101101010000010.01111001100110100_2 \\ &= 2317.0475006103515625. \end{aligned}$$

Signed value representation

When we want to represent a signed value, we have to reserve one bit to handle that sign of the value. For example, this case may occur during the evaluation of a polynomial in fixed-point arithmetic (see Section 5.1, for example). To do so, we consider the integer X in two's complement representation. Hence the real value x encoded by X is defined as follows:

$$x = X \cdot 2^{-f} \quad \text{with} \quad -2^{k-1} \leq X \leq 2^{k-1} - 1.$$

Here f denotes also the number of fraction bits in the binary expansion of x. But in this case the number of bits of the integer part of x is $i = k - f - 1$, since one bit is reserved for the sign. With this representation, we can represent the subset of real values defined as follows:

$$\left\{F \cdot 2^{-f}\right\}_{F=-2^{k-1},\ldots,2^{k-1}-1}.$$

Example 1.4. *Assume $k = 32$. Let X be the 32-bit unsigned integer of Example 1.3:*

$$X = 10110101000001001111001100110100_2.$$

If the integer X encodes the real value in the signed format $Q_{0.31}$, we deduce from $X_{31} = 1$ that it encodes a negative x defined as:

$$\begin{aligned} X = -1257966796 \quad \textit{and} \quad x &= -0.585786437615752220153808593 75 \\ &= -.10010101111101100001100110011002. \end{aligned}$$

Actually, we have $32 - 31 - 1 = 0$ bit of integer part, since $|x| < 1$.

Remark 1.1. *Let X be the* unsigned *binary fixed-point number no larger than $2^{k-1}-1$ that encodes a real positive value x in the format $Q_{i_X.f_X}$. If $i_X \geq 1$, then it follows that X can be seen as a* signed *binary fixed-point number encoding the value x in the format $Q_{i_X\text{-}1.f_X}$, with one sign bit.*

Operations on fixed-point numbers

It remains now to see how to add and multiply two fixed-point numbers. Let us first consider two unsigned binary fixed-point numbers X and Y encoding x in the format Q_{f_x} and y in the format Q_{f_y}, respectively:

$$x = X \cdot 2^{-f_x} \qquad y = Y \cdot 2^{-f_y}, \qquad 0 \leq X, Y < 2^k.$$

Unsigned addition For adding x and y, we first have to scale one of them (y, for example) to align the comma. At the implementation level, this scaling can be simply done by shifting the unsigned integer Y.

Example 1.5. *Assume $k = 32$. Let X and Y be two binary fixed-point numbers defined as:*

$$x = 101101.01000001001111001100110100_2 \quad \textit{and} \quad y = 1.00000010000010010100011001000010_2.$$

We obtain:

$$\begin{array}{lcl} (x) & & 101101.01000001001111001100110100_2 \\ (y) & + & 00000\,1.00000010000010010100011001000010_2 \\ \hline (r) & = & 101110.01000011010001100001001101000010_2 \\ (\hat{r}) & = & 101110.01000011010001100001001101\,00010_2 \end{array}$$

Denoting by $\hat{r}$ the fixed-point number on k bits, we observe on this example that the result $\hat{r}$ is a truncation of the exact result $x + y$, in the format of x.

We observe that addition may entail overflow, or a loss of accuracy while shifting. Moreover, if the comma are already aligned and if no overflow occurs, we remark that addition is error-free.

Unsigned multiplication For multiplying X and Y, we do not have first to align the comma. Hence multiplying two k-bit unsigned binary fixed-point numbers X and Y, in the formats $Q_{i_X.f_X}$ and $Q_{i_Y.f_Y}$, respectively, leads to an exact result R representable on $2k$ bits, in the format $Q_{i_R.f_R}$ with

$$i_R = i_X + i_Y \quad \text{and} \quad f_R = f_X + f_Y.$$

Remark that using the function `mul` defined in (1.1), we can compute the k-bit most significant bits of $X \cdot Y$.

Example 1.6. *Assume $k = 32$. For X and Y as in Example 1.5, we have:*

$$\begin{array}{lcl} (x) & & 101101.01000001001111001100110100_2 \\ (y) & \times & 1.00000010000010010100011001000010_2 \\ \hline (r) & = & 0101101.10011101011000101111111011001100100101001001101000111010000_2 \\ (\hat{r}) & = & 0101101.10011101011000101111111011\,00110010010100100110100011101000_2 \end{array}$$

Denoting by $\hat{r}$ the fixed-point number on k bits encoded by the unsigned integer $\hat{R}$, we observe on this example that the integer $\hat{R}$ can be obtained using the function `mul` *defined in (1.1):*

$$\hat{R} = \texttt{mul}(X, Y).$$

Note that multiplication cannot entail overflow, but a loss of accuracy if we use the function `mul`. Let us bound the error entailed by multiplication. To do so, let $\hat{R}$ be the k most significant bits of the exact product $R = X \cdot Y \cdot 2^{-k}$:

$$0 \leq R - \hat{R} < 1 \quad \text{and} \quad 0 \leq r - \hat{r} < 2^{\mathrm{i}_R - k}. \tag{1.8}$$

It follows that:

$$X \cdot Y \cdot 2^{-k} - 1 < \lfloor X \cdot Y \cdot 2^{-k} \rfloor = \texttt{mul}(X, Y) \leq X \cdot Y \cdot 2^{-k}. \tag{1.9}$$

Remark 1.2. *Concerning the operations on signed binary fixed-point arithmetic, we can make two remarks. Let us now consider X and Y be two signed binary fixed-point numbers encoding x in the format Q_{f_x} and y in the format Q_{f_y}. Then*

- *The addition and multiplication of X and Y are done in the same way as in the case of unsigned fixed-point arithmetic (see [EL04, §1] for details).*
- *The multiplication of two signed binary fixed-point numbers with one sign bit each gives a result with two bits for handling the sign. (See Example 1.7 below, for example.)*

Example 1.7. *Assume $k = 32$. For X as in Example 1.4, we have:*

$$\begin{array}{lcl}
(x) & & -.10010101111101100001100110011001_2 \\
(-x) & \times & +.10010101111101100001100110011001_2 \\
\hline
(r) & = & -.01010111110110000110011001100000000100010101011110000101001_2 \\
(\hat{r}) & = & -.01010111110110000110011001100000001000101010111100001010012
\end{array}$$

with x having 0 bit of integer part and 31 fraction bits. Hence the exact result r has also 0 bit of integer part, but 62 fraction bits.
Denoting by $\hat{r}$ the fixed-point number encoded by the k-bit integer $\hat{R}$ and computed by truncating r on $k-2$ fraction bits, we observe that $\hat{R}$ can be obtained using the function `mul64` *defined in Table 1.1 (and corresponding to the signed version of the function* `mul` *in (1.1)). Since $\hat{R}$ encodes $\hat{r}$ in the signed format $Q_{0.30}$, it follows that*

$$\hat{r} = -0.343145750463008880615234375 \quad \text{and} \quad \hat{R} = -368449944.$$

Hence the bit string of $\hat{R}$ is

$$\hat{R} = \underline{11}10101000001001111001100110100 0_2,$$

where the first two bits encode the sign and indicate that the result is negative. If we had defined $r = x \times x$, we would have obtained

$$\hat{r} = 0.343145750463008880615234375 \quad \text{and} \quad \hat{R} = 368449944.$$

Hence the bit string of $\hat{R}$ would have been

$$\hat{R} = \underline{00}01010111110110000110011001100 0,$$

where the first two bits indicate a positive result.

PART I

Design of optimized algorithms for some binary floating-point operators and their software implementation on embedded VLIW integer processors

CHAPTER 2

Basic blocks for implementing correctly-rounded floating-point operators

This chapter gives a high-level description of the methodology used in this document for implementing correctly-rounded mathematical operators on VLIW integer processors and more particularly brings out various basic blocks useful for these implementations (in particular, the algorithms implemented to deduce this correctly-rounded result from a suitable approximation). For all of them, parametrized descriptions and analyses are proposed, as well as C codes for the standard binary32 *floating-point format optimized for the ST231. To illustrate the interest of this definition and design of basic blocks, a complete C code example is given for the* binary32 *multiplication.*

Until the 80's, each processor constructor had its own floating-point arithmetic implementation, that worked more or less [Mul06], and a numerical code might have different behaviors on different architectures. In 1985, the IEEE 754-1985 standard [Ame85] was published, to standardize and homogenize the implementation, in hardware and software, of binary floating-point arithmetic in processors. One of the main ideas was to ensure that a numerical code has the same behavior on any architecture. At that time, it required in particular correct rounding for the basic arithmetic operators (addition, subtraction, multiplication, and division) and the square root, for four floating-point formats and four rounding modes (round to nearest, toward $+\infty$, toward $-\infty$, toward 0).

This first version has been revised in 2008, and a new version was published on August 29, 2008. In particular, this current version extends the first one to decimal arithmetic and other binary formats; it also specifies the behavior of other mathematical functions like the fused multiply–add (`fma`) or some elementary functions ($\sin$, $\cos$, $\log$, $\exp$,...). More precisely, the IEEE 754-2008 standard requires correct rounding for the four arithmetic operators (addition, subtraction, multiplication, and division), the square root, and the `fma` [IEE08, §5.4.1], while it simply recommends it for the other mathematical functions, such as nth roots, trigonometric functions, logarithms, exponentials, ... [IEE08, §9.2].

One of the goals of this work was to provide an efficient and correctly-rounded support for *binary32* floating-point arithmetic on integer processors, through the correctly-rounded implementation of several mathematical functions: addition, subtraction, and multiplication (see [MBdD+09] for addition and subtraction, and Section 2.4 for multiplication), nth roots (for small, given values

of n) and their reciprocals (Chapter 3), or division (Chapter 4). To define the correctly-rounded value of an exact result, we refer to the IEEE 754-2008 standard [IEE08, pp. 3, 16]: roughly, correct rounding is a method that converts an infinitely precise result to a floating-point number, in a given floating-point format and according to a rounding-direction attribute; this floating-point number is called *correctly-rounded*.

As shown in Figure 2.1 below for univariate function implementations,[1] the computation of the correctly-rounded result $r = f(x)$, with f a mathematical function and x a floating-point datum (as defined in Section 1.2.1), relies usually on several steps. While some of them are function-dependent (in white boxes in Figure 2.1), some others may be implemented once for every function (in grey boxes in Figure 2.1), at least for the functions studied in this thesis. One of the first steps consists in unpacking the floating-point input, as will be described in Section 2.1 for the implementation on VLIW integer processors. Then the IEEE 754-2008 standard requires four rounding-direction attributes [IEE08, §4.3] for binary implementations; and for each of them, Section 2.2 will present the rounding algorithms we have used, when the result is known exactly as well as when it can just be approximated. Section 2.3 will show the method used to construct the correctly-rounded result, and to handle the case where the exact result overflows. Finally Section 2.4 illustrates how to use the basic blocks introduced in this chapter to deduce an implementation for the correctly-rounded *binary32* multiplication. In this chapter, we will not discuss the detection of special input (± 0, $\pm\infty$, NaN) and the selection of special output, which will be detailed further in the document for each kind of function.

2.1 Unpacking an input floating-point number

This section presents the methods used to unpack a floating-point number x encoded into the k-bit unsigned integer X according to the binary interchange encoding defined in Section 1.2.2, and gives some examples of C code for the *binary32* format.

2.1.1 Integer X encoding the binary floating-point number x

Let X be the k-bit unsigned integer encoding of a nonzero (sub)normal floating-point number x, following the binary interchange encoding defined in Section 1.2.2: we have

$$x = (-1)^{s_x} \cdot m_x \cdot 2^{e_x}$$

and

$$X = S_x + E_x \cdot 2^{p-1} + (M_x - n_x \cdot 2^{p-1}),$$

with $S_x = s_x \cdot 2^{k-1}$, $M_x = m_x \cdot 2^{p-1}$, $E_x = e_x - e_{\min} + n_x$, and $X = \sum_{i=0}^{k-1} X_i \cdot 2^i$. Here n_x denotes the "is normal" bit of the floating-point number x: $n_x = 1$ if x is normal, and $n_x = 0$ if x is subnormal. In fact, by definition of m_x in Section 1.2.1, we have $n_x = m_{x,0}$.

Assume that $k = w + p$, as in [IEE08, §3.4]. The bit string of X can be split up into 3 parts: 1 sign bit X_{k-1}, w exponent bits $X_{k-2}, \ldots, X_{p-1}$ such as $E_x = \sum_{i=0}^{w-1} X_{p-1+i} \cdot 2^i$, and $p-1$ fraction bits such as $M_x - n_x \cdot 2^{p-1} = \sum_{i=0}^{p-2} X_i \cdot 2^i$ (see Example 1.1).

2.1.2 Sign and biased exponent extraction

Sign extraction

Extracting of the sign of x is trivial. Let $S_x = s_x \cdot 2^{k-1}$ be the k-bit unsigned integer encoding of the sign of the floating-point number x. Computing S_x may be done by setting the bits $X_{k-2}, \ldots, X_0$

[1] Note that for bivariate functions, Figure 2.1 has to be modified to handle the second variable.

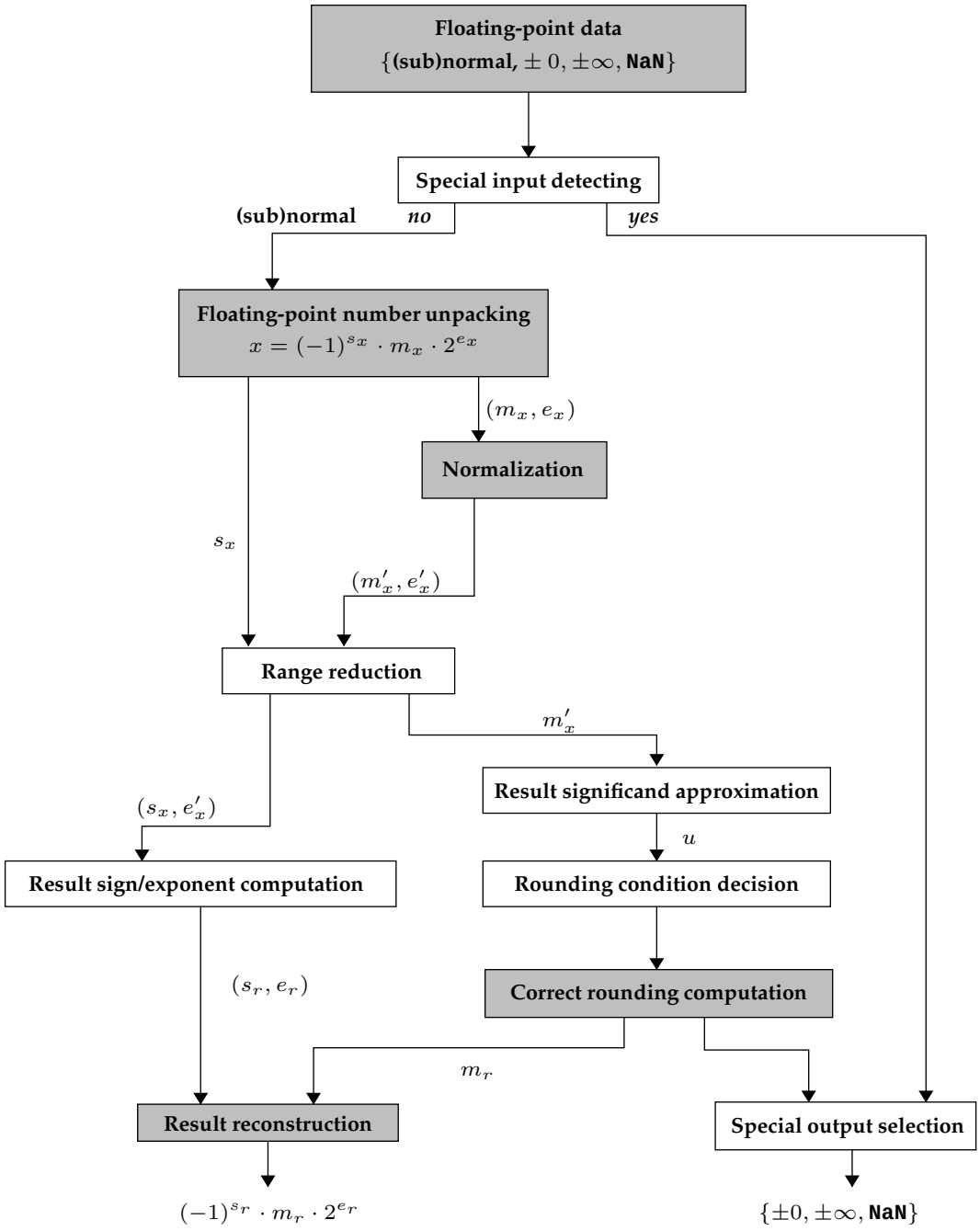


Figure 2.1: Flowchart for correctly-rounded function implementation, for univariate function.

to 0, that is, by taking the bitwise AND between X and $2^{k-1} = 100\cdots00_2$:

$$S_x = X \wedge 2^{k-1}.$$

Exponent extraction

It remains now to explain how to extract from X the integer E_x encoding the biased exponent of the floating-point number x. This may be done by removing the sign bit (setting the bit X_{k-1} to zero) and extracting the following w bits by shifting right the resulting integer by $p-1$ bits. Here, removing the sign bit is done by taking the bitwise AND between X and $2^{k-1}-1 = (011\cdots11)_2$:

$$E_x = \big(X \wedge (2^{k-1}-1)\big) \gg (p-1).$$

Remark that setting X_{k-1} to zero consists in computing the k-bit unsigned integer encoding of the absolute value $|x|$ of x. This integer is denoted here and hereafter by $absX$:

$$E_x = absX \gg (p-1), \quad \text{with} \quad absX = X \wedge (2^{k-1}-1).$$

In C and for $(k, p, e_{\max}) = (32, 24, 127)$, the computation of S_x and E_x can be implemented as presented in lines 1 and 3 of Listing 2.1 below.

```
1  Sx = X & 0x80000000;   absX = X & 0x7FFFFFFF;
2
3  Ex = absX >> 23;       Lx = nlz(absX);  // nlz: leading zero counter
4
5  MX = max(Lx,8);        nx = absX >= 0x00800000;
6
7  Tx = (X << 1) << MX;   Mpx = (X << MX) | 0x80000000;
```

Listing 2.1: Input unpacking implementation for the *binary32* format.

In this piece of code, the variable `nx` encodes n_x, the "is normal" bit of x. Indeed, $n_x = 1$ if and only if $|x| \geq 2^{e_{\min}}$. Since $2^{e_{\min}}$ is encoded by the integer 2^{p-1}, we deduce that $n_x = 1$ if and only if $absX \geq 2^{p-1}$. For $(k, p, e_{\max}) = (32, 24, 127)$, we conclude that $n_x = 1$ if and only if $absX \geq 2^{23} = 0\text{x}00800000_{16}$ (in hexadecimal representation).

As already said, computing E_x could have also been done by first shifting right X by $p-1$ bits, and then removing the bit of weight w storing the sign bit, as follows for $(k, p, e_{\max}) = (32, 24, 127)$:

```
Ex = (X >> 23) & 0xFF;
```

But the advantage of the first solution (Listing 2.1) lies in the fact that the integer $absX$ may be used in other parts of the code, especially for handling special input. If it is not the case, then obviously lines 1 and 3 of Listing 2.1 may also be contracted as follows.

```
Ex = (X & 0x7FFFFFFF) >> 23;
```

Biased exponent extraction, when x is known to be positive

For some functions, the cases $x = 0$ and $x < 0$ are considered as special inputs. As we will observe in Section 3.4 for example, the special input handling is not done from the integer E_x. When in the generic case x is known to be positive, the integer E_x may be extracted simply by right shifting the integer X by $p-1$ bits. In this case, either the sign bit X_{k-1} is already set to 0, or it is equal to 1 and the negative input will be caught up through the special input handling.

This simplifies and eventually may speed up the computation of the biased exponent E_x, as shown below, for $(k, p, e_{\max}) = (32, 24, 127)$:

```
Ex = X >> 23;
```

2.1.3 Normalized significand extraction

In Figure 2.1, one of the first steps of the implementation of a mathematical function consists in normalizing the significand of the input x, in order to get m'_x instead of m_x (see Section 1.2.1). Recall that $m'_x = m_x \cdot 2^{\lambda_x}$, where $\lambda_x \in \{0, \ldots, p-1\}$ is the number of leading zeros in the binary expansion of m_x. We define further its fractional part as

$$\begin{aligned} t_x &= m'_x - 1 \\ &= 0.m_{x,1+\lambda_x} \cdots m_{x,p-1}. \end{aligned}$$

Example 2.1. *Assume* $(k, p, e_{max}) = (32, 24, 127)$. *Let* x_1 *and* x_2 *be two floating-point numbers encoded by, respectively,* X_1 *and* X_2*, such that*

$$X_1 = \boxed{0\ \ 10000001\ \ 01101010000010011110011}$$

and

$$X_2 = \boxed{1\ \ 00000000\ \ 00000000000010011110011}.$$

Here, x_1 *is a* normal *floating-point number, and we have*

$$m'_{x_1} = 1.01101010000010011110011_2 \quad \textit{and} \quad t_{x_1} = 0.\underbrace{01101010000010011110011}_{p-1=23 \textit{ bits}}{}_2.$$

However x_2 *is a* subnormal *floating-point number, and we have*

$$m'_{x_2} = 1.0011110011_2 \quad \textit{and} \quad t_{x_2} = 0.\underbrace{0011110011}_{10 \textit{ bits}}{}_2.$$

Number of leading zeros

Let us first see how to determine the number of leading zeros λ_x. From the definition of the interchange format in Section 1.2.2 and Table 1.4, we know that the trailing significand field of the k-bit integer X carries the bits of m_x (and thus of t_x) as follows:

$$\begin{aligned} X_{p-2} \ldots X_0 &= M_x - n_x \cdot 2^{p-1} \\ &= \underbrace{0 \ldots 01}_{\lambda_x+1 \text{ bits}} m_{x,\lambda_x+1} \ldots m_{x,p-1}. \end{aligned}$$

By definition, we know that a finite x is normal if and only if $E_x \neq 0$, that is, $X_{k-2} \ldots X_{p-1} \neq 0$. Thus let L_x be the number of leading zeros of the integer $absX$ encoding $|x|$. We deduce that:

- $L_x \leq w$ if and only if x is normal and $\lambda_x = 0$,
- $L_x > w$ if and only if x is subnormal and $\lambda_x > 0$.

Thus, when x is subnormal, we further have :

$$L_x = \lambda_x + w.$$

Denoting MX the maximum between the number of leading zeros of $absX$ and the exponent width w, we conclude:

$$\lambda_x = MX - w \qquad \text{with} \qquad MX = \max(Lx, w). \tag{2.1}$$

In fact, in the following, we will not compute λ_x exactly, but only MX, as shown in line 5 of Listing 2.1.

Trailing significand field

Recall that t_x is the trailing significand field of m'_x. From now on, we associate to t_x the following k-bit unsigned integer:

$$T_x = t_x \cdot 2^k.$$

The bits of T_x can be extracted by shifting the integer X left by $MX + 1$ bits. For $(k, p, e_{\max}) = (32, 24, 127)$ and $w = 8$, this may be implemented using the lines 3, 5 and 7 of Listing 2.1. We observe here that we first shift X by 1 bit, and then by MX bits, instead of shifting directly by $MX + 1$ bits. The advantage is twofold:

- It exposes more instruction-level parallelism, and enables to get T_x faster. For example, assuming that `nlz` and `max` have both a latency of 1 cycle, by shifting by $MX + 1$ bits, the value T_x would have been available after 5 cycles, when here, the result T_x is available after 4 cycles.
- Since $|x| \neq 0$ in the generic case, it follows that $absX \neq 0$. The C standard [Int99] requires that $0 \leq MX < k$ for the shift by MX to be well defined: this is the case since $w \leq MX < k$ (with $w > 0$). If we had shifted by $MX + 1$, we could have had an *undefined behavior*, since $MX + 1$ could have been equal to k (when $x = \pm 2^{e_{\min} - p + 1}$, the smallest positive or largest negative subnormal number).

Example 2.2. *Assume $(k, p, e_{\max}) = (32, 24, 127)$. Let x be defined as x_2 in Example 2.1:*

$$X = \boxed{1\ \ 00000000\ \ 01101010000010011110011}\,.$$

Here, we have $L_x = 10$, $MX = 10$, and $\lambda_x = 2$. From the value λ_x, we confirm the fact that x is a subnormal number. Finally, the 32-bit unsigned integer T_x is computed as:

```
Tx = (X << 1)<< 10;
```

and is as follows that

$$T_x = 10101000001001111001100000000000_2,$$

or, in the $Q_{0.32}$ format, $t_x = 0.10101000001001111001 1_2$.

Normalized significand field construction

It remains now to compute the integer M_{px} that encodes m'_x, such as:

$$M_{px} = m'_x \cdot 2^{k-1}.$$

By definition, we know that M_{px} may be computed as $M_{px} = (T_x \cdot 2^{-1}) + 2^{k-1}$. Note that here the division by 2 is exact. Assuming $(k, p, e_{\max}) = (32, 24, 127)$, the implementation is thus as follows.

```
Mpx = (Tx >> 1) + 0x80000000;
```

Here, we have to wait for the value T_x before starting the computation of M_{px}. Observe in Listing 2.1 that the computation of the value MX is cheaper than that of T_x. It follows that M_{px} may also be computed faster directly by shifting X left by MX bits, and forcing the bit of weight $k-1$, representing the bit m_{x,λ_x} (the integer part of m'_x), to be 1. This may be implemented as at line 7 of Listing 2.1.

Simplifications when subnormal numbers are not supported

Until now, we have considered that x can be a subnormal floating-point number. However, if subnormal numbers are not supported,[2] the implementation can of course be simplified, especially the extraction of the trailing significand field and the construction of the normalized significand. Indeed, in this case, $(MX, \lambda_x) = (w, 0)$, and extracting the bits of the trailing significand may be done by shifting the integer X left by $w+1$ bits. For $(k, p, e_{\max}) = (32, 24, 127)$ and $w = 8$, this can be implemented in C as follows.

```
Tx = X << 9;

Mpx = (X << 8) | 0x80000000;
```

2.2 Implementation of rounding algorithms

This section defines the correctly-rounded value of a real number and the algorithms that we will use in the sequel to compute it. It also gives some key elements for implementing these rounding algorithms on VLIW integer processors.

2.2.1 Correctly-rounded value of a real number

Let r be a nonzero real number.[3] This section defines the correctly-rounded value of r, denoted by $\circ(r)$. First we choose to express that real value r as follows:

$$r = (-1)^{s_r} \cdot \ell \cdot 2^d, \tag{2.2}$$

with $s_r \in \{0, 1\}$, $\ell \in \mathbb{R} \cap [1, 2]$, and $d \in \mathbb{Z}$. Remark that the closed interval $[1, 2]$ comes from the fact that for some functions, the real value ℓ lies in the range $[1, 2)$, while for others it lies in $(1, 2]$. Here, we consider three cases:

- either $d > e_{\max}$ and the exact result r overflows since then $|r| > \Omega$;
- or $e_{\min} \leq d \leq e_{\max}$ and the exact result r lies in the range of normal floating-point numbers (or $\Omega < |r| \leq 2^{e_{\max}+1}$ when $2 - 2^{1-p} < \ell \leq 2$ and $d = e_{\max}$ [4]);
- or $d < e_{\min}$ and the exact result r lies in the range of subnormal floating-point numbers (or is equal to $2^{e_{\min}}$ when $\ell = 2$ and $d = 2^{e_{\min}-1}$).

[2] We consider in this case that an input cannot be a subnormal floating-point number.

[3] Recall that ± 0 is considered as a special input and its handling will be described further in the document.

[4] If $2 - 2^{1-p} < \ell \leq 2$ and $d = e_{\max}$ then the exact result r is outside the normal range, but this case will be detailed in Section 2.3.1.

To handle all these cases *simultaneously*, we define the following integer λ_r:

$$\lambda_r = \max(0, e_{\min} - d). \tag{2.3}$$

Defining further

$$e_r = \max(e_{\min}, d), \tag{2.4}$$

we get clearly

$$r = (-1)^{s_r} \cdot (\ell \cdot 2^{-\lambda_r}) \cdot 2^{e_r}. \tag{2.5}$$

and

$$\circ(r) = (-1)^{s_r} \cdot m_r \cdot 2^{e_r}, \quad \text{with} \quad m_r = \circ(\ell \cdot 2^{-\lambda_r}). \tag{2.6}$$

The case $d > e_{\max}$ is explained in more details in Section 2.3.2. Remark that the maximal value for λ_r depends on the function. Therefore we simply ensure here that:

- if $d \in \{e_{\min}, \ldots, e_{\max}\}$ then $\lambda_r = 0$ and $\ell \in [1, 2]$;
- otherwise $\lambda_r > 0$ and $\ell \cdot 2^{-\lambda_r} \in (0, 1]$.

Hence m_r lies in $[1, 2]$ or $[0, 1]$, and may not correspond to the significand of the correctly-rounded floating-point result $\circ(r)$.

From now, we observe that the case $m_r = 2$ may occur, and if furthermore $e_r = e_{\max}$ then the function overflows. We will see in Section 2.3.1 that this special case can be handled through the rounding procedure, or as a special implementation case (see for example Section 3.6.3 for an illustration in the case of the reciprocal function $x \mapsto 1/x$), depending on the rounding-direction attribute.

2.2.2 Rounding-direction attributes

For radix 2, the IEEE 754-2008 standard requires four rounding-direction attributes [IEE08, §4.3]: RoundTiesToEven (RN_p), RoundTowardPositive (RU_p), RoundTowardNegative (RD_p), and RoundTowardZero (RZ_p), where p denotes the precision of the binary floating-point format. A fifth one, RoundTiesToAway, is simply recommended by the IEEE 754-2008 standard [IEE08, §4.3.3] for binary implementation, but has not been implemented in FLIP. Figure 2.2 illustrates the relationship between r and $\circ(r)$ in (2.6) for each of the four required rounding-direction attributes.

In particular, we observe for RoundTiesToEven and RoundTowardZero that

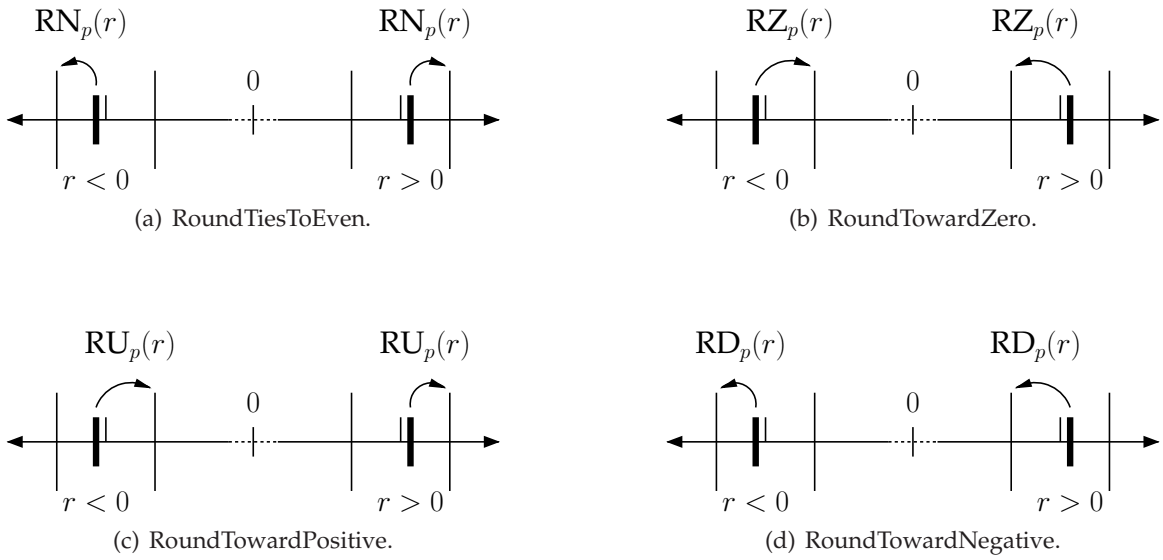
$$\mathrm{RN}_p(-r) = -\mathrm{RN}_p(r) \quad \text{and} \quad \mathrm{RZ}_p(-r) = -\mathrm{RZ}_p(r),$$

and for these two rounding-direction attributes, the sign of the result does not affect the rounding algorithm and its implementation, that is, we can restrict to $|r|$, the absolute value of r. However for RoundTowardPositive and RoundTowardNegative, we have

$$\mathrm{RU}_p(-r) = -\mathrm{RD}_p(r) \quad \text{and} \quad \mathrm{RD}_p(-r) = -\mathrm{RU}_p(r),$$

and for these two rounding-direction attributes, the rounding algorithms depend on the sign s_r of the exact result r.

From Figure 2.1, for implementing a function, an objective is to compute the correctly-rounded significand $m_r = \circ(\ell \cdot 2^{-\lambda_r})$ in parallel with the pair (result sign, exponent), and to construct the final result by combining these three elements. Using the definitions of r and $\circ(r)$ in (2.6), we can deduce Table 2.1 below, which shows the relationship required between $\ell \cdot 2^{-\lambda_r}$ and m_r to ensure correct rounding.



(a) RoundTiesToEven. (b) RoundTowardZero.

(c) RoundTowardPositive. (d) RoundTowardNegative.

Figure 2.2: Rounding-direction attributes.

Rounding-direction attribute	Relationship between $\ell \cdot 2^{-\lambda_r}$ and m_r
RoundTiesToEven	$\lvert \ell \cdot 2^{-\lambda_r} - m_r \rvert \leq 2^{-p}$
RoundTowardPositive	$2^{1-p} < (-1)^{s_r} \cdot (\ell \cdot 2^{-\lambda_r} - m_r) \leq 0$
RoundTowardNegative	$0 \leq (-1)^{s_r} \cdot (\ell \cdot 2^{-\lambda_r} - m_r) < 2^{1-p}$
RoundTowardZero	$0 \leq \ell \cdot 2^{-\lambda_r} - m_r < 2^{1-p}$

Table 2.1: Relationship between $\ell \cdot 2^{-\lambda_r}$ and m_r, for $\circ \in \{\text{RN}_p, \text{RU}_p, \text{RD}_p, \text{RZ}_p\}$.

2.2.3 When the exact result is finite

This section presents the algorithms used for implementing correct rounding when the exact result r has a finite number of bits and thus can be computed exactly, as presented in [EL04, pp. 436–437] for multiplication. Recall that $\ell \cdot 2^{-\lambda_r}$ in (2.6) represents the scaled significand of the exact result r to be rounded into $m_r = \circ(\ell \cdot 2^{-\lambda_r})$. Recall also that $\ell \in [1, 2]$. We assume in this section that $\ell \cdot 2^{-\lambda_r}$ has a finite binary expansion, let say in f fraction bits:

$$\ell \cdot 2^{-\lambda_r} = \underbrace{00.0\ldots0}_{\lambda_r \text{ bits}} \ell_{-1}\ell_0\ell_1\ell_2\ldots\ell_{f-\lambda_r}.$$

In our context, the implementation of mathematical functions is done using k-bit numbers, with $k > p-2$ (which is the case for each of the binary floating-point formats defined by the IEEE 754-2008 standard). In this document, we denote by $\texttt{truncate}(n)_t$ the truncation after t fraction bits of the value n, such as:

$$\texttt{truncate}(n)_t = \lfloor n \cdot 2^t \rfloor / 2^t. \tag{2.7}$$

Consequently, let u be the value of $\ell \cdot 2^{-\lambda_r}$ truncated after $k-2$ fraction bits:

$$\begin{aligned} u &= \texttt{truncate}(\ell \cdot 2^{-\lambda_r})_{k-2} \\ &= u_{-1}u_0.u_1\ldots u_{p-1}u_pu_{p+1}\ldots u_{k-2} \\ &= \underbrace{00.0\ldots0}_{\lambda_r \text{ bits}} \ell_{-1}\ell_0\ell_1\ell_2\ldots\ell_{k-2-\lambda_r}. \end{aligned} \tag{2.8}$$

Furthermore, define a *guard bit* g and a *sticky bit* s as follows:

$$g = u_p = \ell_{p-\lambda_r} \quad \text{and} \quad s = \text{“logical or” of the bits } \ell_i \text{ for } i \in \{p-\lambda_r+1, \ldots, f-\lambda_r\}. \tag{2.9}$$

According to those two bits and the considered rounding-direction attribute, as presented in [EL04, pp. 436–437] the rounding procedure consists then in determining a *rounding bit* b such that,

$$m_r = \texttt{truncate}(u)_{p-1} + b \cdot 2^{1-p} \tag{2.10}$$

satisfies one of the conditions in Table 2.1 (depending on the chosen rounding-direction attribute). Assume now that U and M_r are k-bit unsigned integers that encode u and m_r, respectively, as

$$U = u \cdot 2^{k-2} \quad \text{and} \quad M_r = m_r \cdot 2^{p-1}. \tag{2.11}$$

Then, multiplying both sides of (2.10) by 2^{p-1} and using (2.7), we obtain:

$$M_r = \lfloor U/2^{k-p-1} \rfloor + b.$$

In integer arithmetic, the computation of $\lfloor U/2^{k-p-1} \rfloor$ simply consists in shifting U right by $k-p-1$ bits:

$$M_r = \big(U \gg (k-p-1)\big) + b.$$

Hence, assuming for example that $(k, p, e_{\max}) = (32, 24, 127)$ and given the integers U and b, the computation of M_r can be done with the following line of C code:

```
Mr = (U >> 7) + b;
```

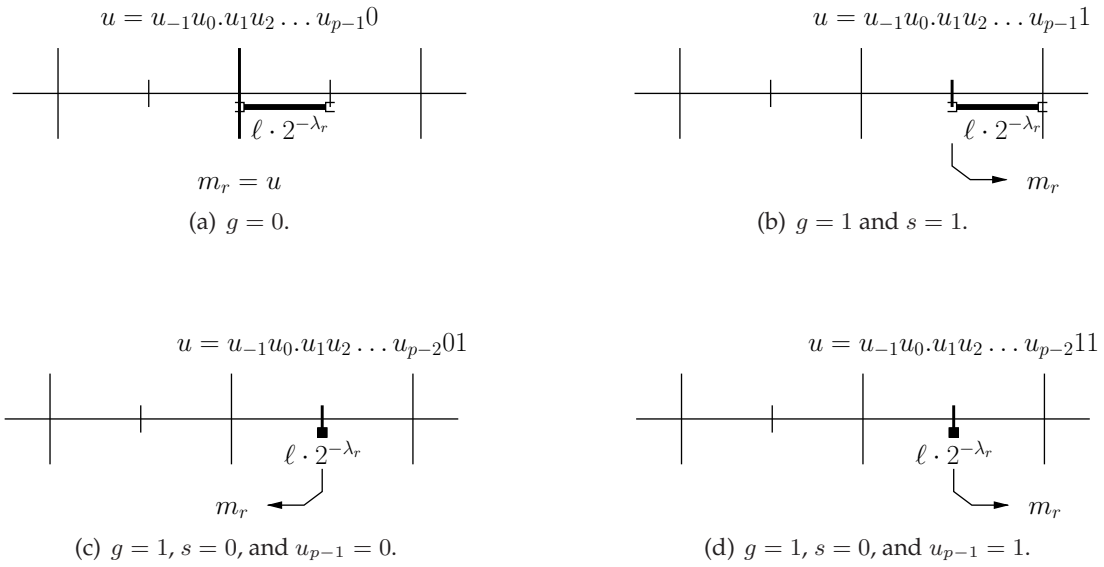
Once the integer M_r is known, the final result will follow from the packing procedure we shall describe in Section 2.3.

Now, the problem is from u, g, and s to determine the rounding bit b. In the remainder of this section, we describe the computation of b for each rounding-direction attribute. Since the method used to compute u, g, and s is function-dependent, it will simply be detailed on an example (multiplication) in Section 2.4.

Computing b for RoundTiesToEven

From Table 2.1, in RoundTiesToEven, we know that the rounded significand m_r is defined by $\left|\ell \cdot 2^{-\lambda_r} - m_r\right| \leq 2^{-p}$, and if a tie occurs (that is, $\ell \cdot 2^{-\lambda_r}$ is exactly between two floating-point numbers) then the IEEE 754-2008 standard requires that the least significant bit of m_r be zero (that is, m_r has an even significand). This is illustrated in Figure 2.3 below. Hence, the rounding bit b can be computed from g, s, and u_{p-1} as follows:

$$b = \begin{cases} 0, & \text{if } g = 0, \\ 1, & \text{if } g = 1 \text{ and } s = 1, \\ u_{p-1}, & \text{if } g = 1 \text{ and } s = 0. \end{cases} \tag{2.12}$$



(a) $g = 0$. (b) $g = 1$ and $s = 1$.

(c) $g = 1$, $s = 0$, and $u_{p-1} = 0$. (d) $g = 1$, $s = 0$, and $u_{p-1} = 1$.

Figure 2.3: RoundTiesToEven for $r > 0$.

Indeed, if $g = 0$ (Figure 2.3(a)) then $m_r = \texttt{truncate}(u)_{p-1}$ in (2.10), and if $g = 1$ and $s = 1$ (Figure 2.3(b)) then $m_r = \texttt{truncate}(u)_{p-1} + 2^{1-p}$: in both cases, it is the expected result. If $g = 1$ and $s = 0$, the value u is exactly halfway between two floating-point numbers. In this case, the expected result is the floating-point number having even significand, which is the returned result: if $u_{p-1} = 0$ (Figure 2.3(c)) then $m_r = \texttt{truncate}(u)_{p-1}$, and if $u_{p-1} = 1$ (Figure 2.3(d)) then $m_r = \texttt{truncate}(u)_{p-1} + 2^{1-p}$. From the definition of b in (2.12), we can deduce the following boolean formula

$$b = g \wedge (u_{p-1} \vee s).$$

Getting the bit u_{p-1} from U in (2.11) may be done by shifting U right by $k - p - 1$ bits, and then removing the first $k - 1$ bits of the resulting integer:

$$u_{p-1} = \big(U \gg (k - p - 1)\big) \wedge 1.$$

For $(k, p, e_{\max}) = (32, 24, 127)$, this may be implemented in C as follows, with `g` and `s` two k-bit unsigned integers encoding g and s, respectively.

```
u_p_1 = (U >> 7) & 1;

b = g & (u_p_1 | s);
```

Example 2.3. *Assume* $(k, p, e_{max}) = (32, 24, 127)$. *Let* $\lambda_r = 0$ *and let* ℓ *be the exact significand of a real number* r*, whose binary expansion is exactly representable on* 48 *bits:*

$$\ell = 01.\underbrace{111111111111111111111}_{p-1=23 \text{ bits}}\underline{0}11011001111110101010012\,.$$

Here, we have the guard bit $g = 0$ *(underlined in the above binary expansion) and the sticky bit* $s = 1$*. Thus, from* (2.12) *we deduce that the rounding bit is* $b = 0$ *and that*

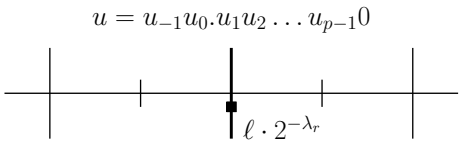
$$\begin{aligned} m_r = \circ(\ell) &= 01.11111111111111111111112 + 0 \cdot 2^{-23} \\ &= 01.11111111111111111111112\,, \end{aligned}$$

which is actually the number with 23 *fraction bits that is closest to* ℓ*.*

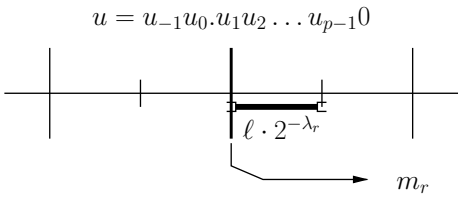
Computing b for RoundTowardPositive

In RoundTowardPositive, we see from Table 2.1 that the resulting m_r has to satisfy the following condition: $2^{1-p} < (-1)^{s_r} \cdot (\ell \cdot 2^{-\lambda_r} - m_r) \leq 0$. Assume first that the result r is positive, that is, $s_r = 0$. In this case the rounding bit b can be computed from the bits of u as follows:

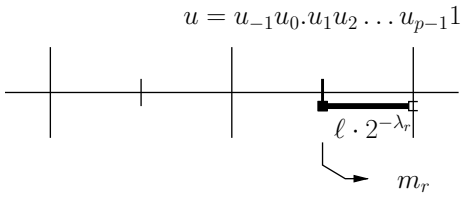
$$b = \begin{cases} 0 & \text{if } g = 0 \text{ and } s = 0, \\ 1 & \text{otherwise.} \end{cases} \tag{2.13}$$



(a) $g = 0$ and $s = 0$.



(b) $g = 0$ and $s = 1$.



(c) $g = 1$.

Figure 2.4: RoundTowardPositive for $r > 0$.

This means that if $g = 0$ and $s = 0$ (Figure 2.4(a)), the value u is already the significand of a floating-point number, and $m_r = \texttt{truncate}(u)_{p-1}$ in (2.10). Otherwise, the exact result is strictly between two consecutive floating-point numbers (Figure 2.4(b) and Figure 2.4(c)), and $m_r = \texttt{truncate}(u)_{p-1} + 2^{1-p}$, which is the floating-point number above. Assume now that the result is negative, that is, $s_r = 1$. Since $\mathrm{RU}_p(-r) = -\mathrm{RD}_p(r)$, the rounding bit is always 0 (see the next paragraph for details on computing b for RoundTowardNegative). In summary, for

RoundTowardPositive, the rounding bit b is given by the following formula

$$b = \neg s_r \wedge (g \vee s).$$

Since S_r is the k-bit unsigned integer that encodes s_r as $s_r = S_r \cdot 2^{1-k}$, for $(k, p, e_{\max}) = (32, 24, 127)$ the computation of b can be implemented in C using the following piece of code.

```
b = (!(Sr >> 31)) & (g | s);
```

Example 2.4. *Assume $(k, p, e_{\max}) = (32, 24, 127)$. Let $s_r = 0$ and let λ_r, ℓ be defined as in Example 2.3:*

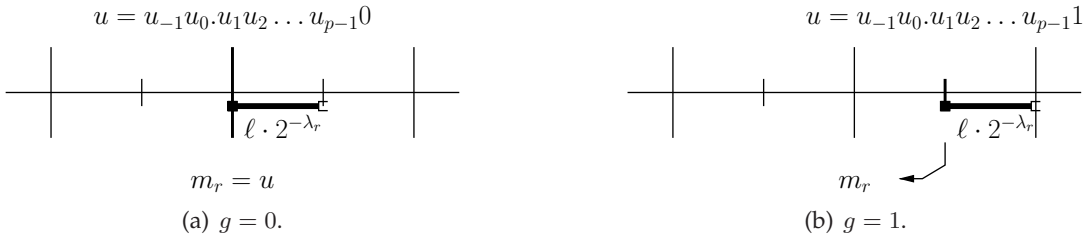
$$\ell = 01.\underbrace{11111111111111111111111}_{p-1=23 \text{ bits}}0110110011111101010100 1_2.$$

Recall that the guard bit is $g = 0$ (underlined in the above binary expansion) and that the sticky bit is $s = 1$. Thus, from (2.13) we deduce that the rounding bit is $b = 1$ and that

$$\begin{aligned} m_r &= 01.11111111111111111111111_2 + 1 \cdot 2^{-23} \\ &= 10.00000000000000000000000_2. \end{aligned}$$

Computing b for RoundTowardNegative

From Table 2.1, for RoundTowardNegative, the value m_r has to satisfy $0 \leq (-1)^{s_r} \cdot (\ell \cdot 2^{-\lambda_r} - m_r) < 2^{1-p}$. Here, assume again first that the result r is positive, that is, $s_r = 0$.



(a) $g = 0$. (b) $g = 1$.

Figure 2.5: RoundTowardNegative for $r > 0$.

In this case, the computation is trivial since $\texttt{truncate}(u)_{p-1}$ is already the significand of $\ell \cdot 2^{-\lambda_r}$ rounded downward, and whatever the value of the bits g and s are, the expected result is $\texttt{truncate}(u)_{p-1}$. This is illustrated in Figure 2.5 above. Hence the rounding bit b is always 0. Assume now that $s_r = 1$ (the result is negative). Since $\mathrm{RD}_p(-r) = -\mathrm{RU}_p(r)$, the rounding bit b has to be equivalent to the one defined previously for RoundTowardPositive when $s_r = 0$. It follows that the rounding bit is given by

$$b = s_r \wedge (g \vee s). \tag{2.14}$$

Its implementation for $(k, p, e_{\max}) = (32, 24, 127)$ may be derivated from the one defined above for the RoundTowardPositive, and the piece of C code below follows.

```
b = (Sr >> 31) & (g | s);
```

Example 2.5. *Assume $(k, p, e_{\max}) = (32, 24, 127)$. Let $s_r = 0$ and let λ_r, ℓ be defined as in Example 2.3:*

$$\ell = 01.\underbrace{11111111111111111111111}_{p-1=23 \text{ bits}}0110110011111101010100 1_2.$$

Recall that the guard bit is $g = 0$ (underlined in the above binary expansion) and that the sticky bit is $s = 1$. Thus, from (2.14) we deduce that the rounding bit is $b = 0$ and that

$$\begin{aligned} m_r &= 01.11111111111111111111111_2 + 0 \cdot 2^{-23} \\ &= 01.11111111111111111111111_2. \end{aligned}$$

Computing b for RoundTowardZero

For the last rounding-direction attribute, from Table 2.1, the returned value m_r has to satisfy: $0 \leq \ell \cdot 2^{-\lambda_r} - m_r < 2^{1-p}$. This rounding-direction attribute can be considered as the simplest to implement, since it does not depend on the sign of the result and $\texttt{truncate}(u)_{p-1}$ is already the expected output. Thus the rounding bit is always $b = 0$, and its implementation in C is trivial:

```
b = 0;
```

2.2.4 When the exact result may have an infinite number of digits

This section presents the algorithms used to compute the correct rounding of an exact result r that may have an infinite binary expansion. Indeed, in this case, the *sticky bit* in (2.9) cannot always be computed in a finite amount of time, and the rounding bit cannot be determined as described in Section 2.2.3. Recall that we want to compute the correctly-rounded result r defined in (2.6) as $\circ(r) = (-1)^{s_r} \cdot m_r \cdot 2^{e_r}$, with $m_r = \circ(\ell \cdot 2^{\lambda_r})$. To ensure correct rounding, in this case, the main idea consists in computing an approximation u of the real value $\ell \cdot 2^{-\lambda_r}$ exactly representable on p fraction bits: $u = u_{-1}u_0.u_1u_2 \dots u_p$, such as

$$|\ell \cdot 2^{-\lambda_r} - u| < 2^{-p}. \tag{2.15}$$

In our context, computing such an approximation u will rely on the computation of a *one-sided approximation* v of $\ell \cdot 2^{-\lambda_r}$, that approximates the real value from above, such as $-2^{-p} < \ell \cdot 2^{-\lambda_r} - v \leq 0$, as in [EL04]. It follows that the value u may be obtained by truncating v after p fraction bits: indeed truncation gives $0 \leq v - u < 2^{-p}$ and (2.15) holds. (This will be explained later on in more details for each of the functions studied in this thesis; see Chapters 3 and 4.)

Now we can distinguish two cases: either u is the significand of a floating-point number representable on $p - 1$ bits (Figure 2.6(a)), or it is halfway between two consecutive floating-point numbers (Figure 2.6(b)).

(a) $u_p = 0$. (b) $u_p = 1$.

Figure 2.6: Relationship between $\ell \cdot 2^{-\lambda_r}$ and u, when ℓ is approximated from above.

Given the values u and $\ell \cdot 2^{-\lambda_r}$ and a rounding-direction attribute, recall from (2.6) that the correctly-rounded result $\circ(r)$ is defined as:

$$\circ(r) = (-1)^{s_r} \cdot m_r \cdot 2^{e_r}, \quad \text{with} \quad m_r = \circ(\ell \cdot 2^{\lambda_r}).$$

Here, obtaining the correctly-rounded significand m_r relies on determining if the "rounding condition" $u \geq \ell \cdot 2^{-\lambda_r}$ or $u > \ell \cdot 2^{-\lambda_r}$, depending on the rounding-direction attribute, is true. In

this section, we do not discuss the implementation of this rounding test: we will consider in the following that the variable `cond` is equal to 1 (true) if and only if the rounding test is satisfied, and 0 (false) otherwise. Its implementation will be detailed in Chapters 3 and 4, depending on the function considered.

In the remainder of this section, we describe the rounding algorithm used to deduce m_r and give some key elements for constructing the integer result M_r from U, for each rounding-direction attribute. As in the previous section, we assume the following definition for U and M_r:

$$U = u \cdot 2^{k-2} \quad \text{and} \quad M_r = m_r \cdot 2^{p-1}. \tag{2.16}$$

Recall also that $\texttt{truncate}(n)_t$ denotes truncation after t fraction bits of the real value n:

$$\texttt{truncate}(n)_t = \lfloor n \cdot 2^t \rfloor / 2^t.$$

Computing m_r for RoundTiesToEven

Recall that from Table 2.1 in RoundTiesToEven, we want to compute the floating-point number m_r of even significand such that $\left| m_r - \ell \cdot 2^{-\lambda_r} \right| \leq 2^{-p}$.

Rounding algorithm 2.1 (RoundTiesToEven)**.** *In RoundTiesToEven, the floating-point m_r can be computed as follows:*

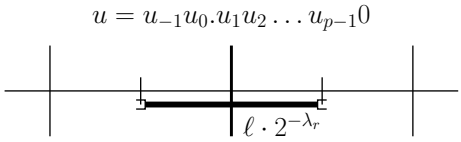
if $u > \ell \cdot 2^{-\lambda_r}$ *or* $\left(u = \ell \cdot 2^{-\lambda_r}$ *and* $u_{p-1} = 0\right)$ *then*
$\quad m_r = \texttt{truncate}(u)_{p-1}$
else
$\quad m_r = \texttt{truncate}(u + 2^{-p})_{p-1}$.

Proof. Since u is exactly representable on p fraction bits, if $u_p = 0$ this algorithm always returns the value u. This is the required result, since u is already the significand of a floating-point number with $p-1$ fraction bits (Figure 2.7(a)). If $u_p = 1$, the value u is exactly halfway between the two consecutive floating-point numbers $u - 2^{-p}$ and $u + 2^{-p}$: the former is returned if $u > \ell \cdot 2^{-\lambda_r}$ or if $u = \ell \cdot 2^{-\lambda_r}$ and $u_{p-1} = 0$ (Figure 2.7(b)), the latter is returned if $u < \ell \cdot 2^{-\lambda_r}$ or if $u = \ell \cdot 2^{-\lambda_r}$ and $u_{p-1} = 1$ (Figure 2.7(c)), that are both the expected result. Here, if $\ell \cdot 2^{-\lambda_r}$ is a midpoint between two consecutive floating-point numbers, m_r has an even significand has required in [IEE08]. □

It remains now to explain how to deduce the integer M_r from the integer U in (2.16). Consider the case $m_r = \texttt{truncate}(u + 2^{-p})_{p-1}$. We deduce that $M_r = \texttt{truncate}(u + 2^{-p})_{p-1} \cdot 2^{p-1}$, and thus, multiplying both sides by 2^{p-1}, and using (2.16) and the definition of truncation in (2.7), we obtain $M_r = \lfloor (U + 2^{k-p-2})/2^{k-p-1} \rfloor$. Similarly, $M_r = \texttt{truncate}(u)_{p-1} \cdot 2^{p-1} = \lfloor U/2^{k-p-1} \rfloor$. In integer arithmetic $\lfloor U/2^i \rfloor$ is easily computable using a right shift by i bits. Consequently, for $(k, p, e_{\max}) = (32, 24, 127)$, the following piece of C code implements the rounding procedure and computes the integer M_r encoding the correctly-rounded significand m_r.

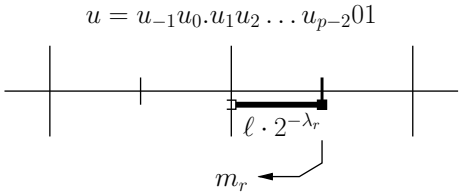
```
if(cond){
  Mr = U >> 7;
}else{
  Mr = (U + 0x40) >> 7;
}
```

As we have seen above in introduction, here we consider that `cond` is true if and only if the rounding condition $\left(u > \ell \cdot 2^{-\lambda_r}$ or $\left(u = \ell \cdot 2^{-\lambda_r}$ and $u_{p-1} = 0\right)\right)$ is true. We will study in Chapters 3 and 4 how to implement this rounding test, depending on the function being implemented.

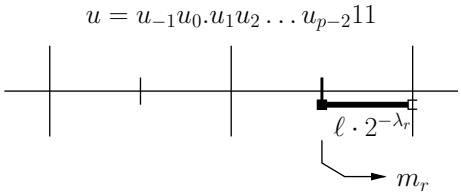


$m_r = u$

(a) $u_p = 0$.



(b) $u_{p-1} = 0$ and $u_p = 1$.



(c) $u_{p-1} = 1$ and $u_p = 1$.

Figure 2.7: RoundTiesToEven for $r > 0$.

Computing m_r for RoundTowardPositive

In RoundTowardPositive, the correctly-rounded result depends on the sign of the result s_r, as follows: $\mathrm{RU}_p(-r) = -\mathrm{RD}_p(r)$, where r denotes the exact result. From Table 2.1, the goal consists here in computing the floating-point number m_r such that $-2^{1-p} < (-1)^{s_r} \cdot (\ell \cdot 2^{-\lambda_r} - m_r) \leq 0$. Assume first that the exact result is positive, that is, $s_r = 0$.

Rounding algorithm 2.2 (RoundTowardPositive). *Assuming $r > 0$, in RoundTowardPositive, the computation of the floating-point number m_r (such as $-2^{1-p} < \ell \cdot 2^{-\lambda_r} - m_r \leq 0$) can be done as follows:*

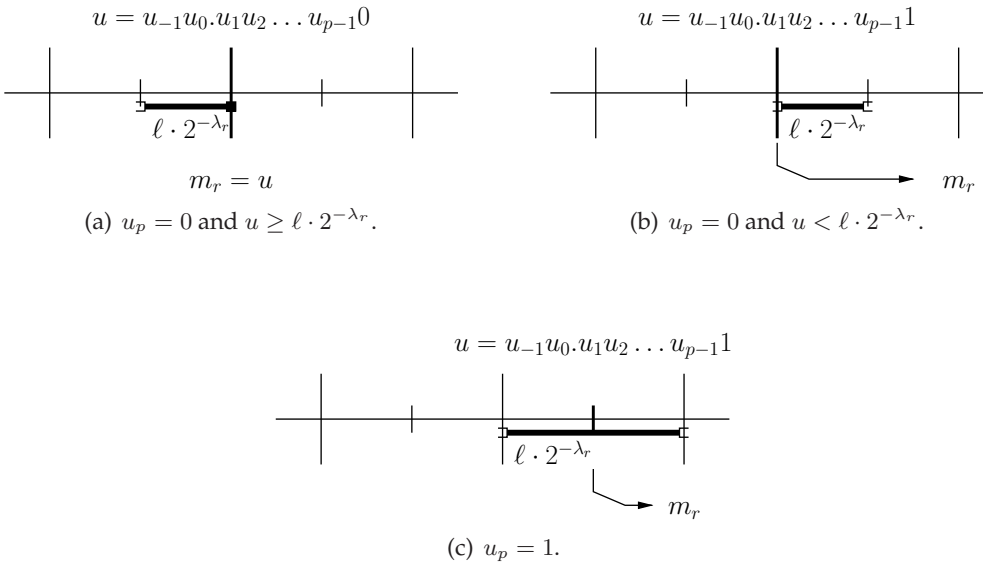
if $u \geq \ell \cdot 2^{-\lambda_r}$ *then*
$$m_r = \texttt{truncate}(u + 2^{-p})_{p-1}$$
else
$$m_r = \texttt{truncate}(u + 2^{1-p})_{p-1}.$$

Proof. Recall that u has a finite representation on p fraction bits. If $u_p = 1$ then the value u is halfway between the two floating-point numbers $u - 2^{-p}$ and $u + 2^{-p}$, and this algorithm always returns $u + 2^{-p}$, which is the required result (Figure 2.8(c)). If $u_p = 0$ then u is already the significand of a floating-point number. If $u \geq \ell \cdot 2^{-\lambda_r}$ then $m_r = u$ (Figure 2.8(a)) and if $u < \ell \cdot 2^{-\lambda_r}$ then $m_r = u + 2^{1-p}$ (Figure 2.8(b)). In both cases the algorithm returns the expected result. □

Following the same approach as for RoundTiesToEven, we deduce that the resulting integer is defined as:

$$M_r = \begin{cases} \lfloor (U + 2^{k-p-2})/2^{k-p-1} \rfloor & \text{if } u \geq \ell \cdot 2^{-\lambda_r}, \\ \lfloor (U + 2^{k-p-1})/2^{k-p-1} \rfloor & \text{otherwise.} \end{cases} \tag{2.17}$$

For $(k, p, e_{\max}) = (32, 24, 127)$, the following piece of C code implements the rounding procedure, using the formulae in (2.17) and for $S_r = s_r = 0$:



(a) $u_p = 0$ and $u \geq \ell \cdot 2^{-\lambda_r}$.

(b) $u_p = 0$ and $u < \ell \cdot 2^{-\lambda_r}$.

(c) $u_p = 1$.

Figure 2.8: RoundTowardPositive for $r > 0$.

```
if(cond){
  Mr = (U + (0x40 - (Sr >> 24))) >> 7;
}else{
  Mr = (U + (0x80 - (Sr >> 24))) >> 7;
}
```

In fact, the above code works for $s_r = 1$ as well, since in this case we want $m_r = \mathrm{RD}_p(\ell \cdot 2^{-\lambda_r})$. Indeed we may check that if the result is negative then $S_r = 2^{k-1}$, and it follows that:

$$U - 2^{k-p-2} = U + 2^{k-p-2} - S_r \cdot 2^{-p},$$

and

$$U = U + 2^{k-p-1} - S_r \cdot 2^{-p},$$

that will both be used in the next paragraph to define the integer M_r in (2.18) for RoundToward-Negative. Hence, when $s_r = 1$ this code actually implements the rounding algorithm presented below for RoundTowardNegative and $s_r = 0$.

Computing m_r for RoundTowardNegative

From Table 2.1, in RoundNowardNegative, we also have to consider the sign of the result to handle the correctly-rounded value m_r, such as $0 \leq (-1)^{s_r} \cdot (m_r - \ell \cdot 2^{-p}) < 2^{1-p}$. When $s_r = 0$, we can use the following algorithm.

Rounding algorithm 2.3 (RoundTowardNegative). *Assuming $r > 0$, in RoundTowardNegative, the floating-point number m_r can be computed as follows:*

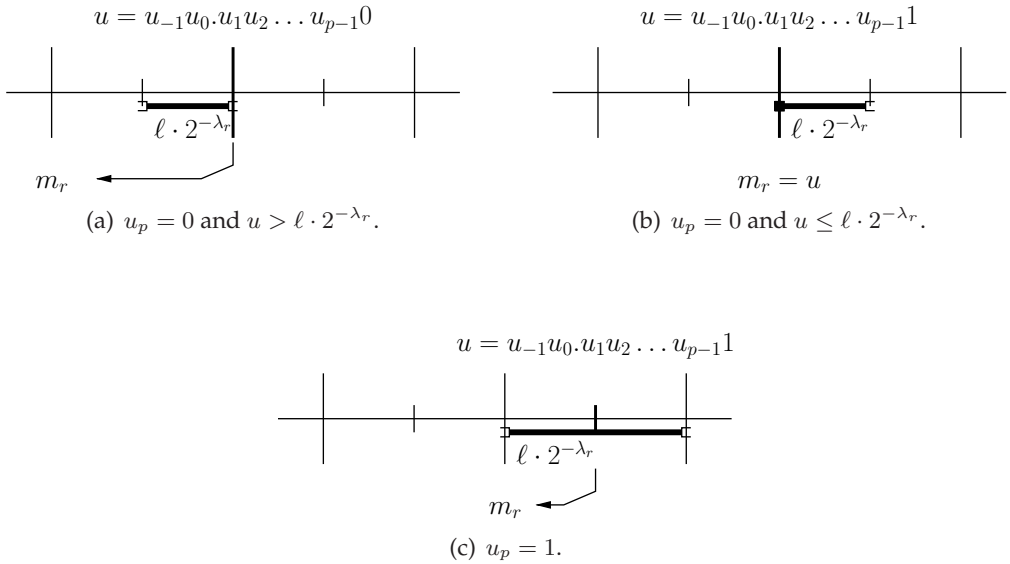
if $u > \ell \cdot 2^{-\lambda_r}$ *then*

$$m_r = \texttt{truncate}(u - 2^{-p})_{p-1}$$

else

$$m_r = \texttt{truncate}(u)_{p-1}.$$

Proof. The value u having a finite binary expansion on p fraction bits, if $u_p = 1$ then the value u is exactly halfway between the two consecutive floating-point numbers $u - 2^{-p}$ and $u + 2^{-p}$, and this algorithm always returns $u - 2^{-p}$, which is the expected result (Figure 2.9(c)). If $u_p = 0$ then u is already the significand of a floating-point number: if $u > \ell \cdot 2^{-\lambda_r}$ then $m_r = u - 2^{1-p}$ (Figure 2.9(a)), and if $u \leq \ell \cdot 2^{-\lambda_r}$ then $m_r = u$ (Figure 2.9(b)); and in both cases m_r is the required result. □



(a) $u_p = 0$ and $u > \ell \cdot 2^{-\lambda_r}$.

(b) $u_p = 0$ and $u \leq \ell \cdot 2^{-\lambda_r}$.

(c) $u_p = 1$.

Figure 2.9: RoundTowardNegative for $r > 0$.

Still using the same approach as for previous rounding-direction attribute, we deduce that the resulting integer M_r is as follows:

$$M_r = \begin{cases} \lfloor (U - 2^{k-p-2})/2^{k-p-1} \rfloor & \text{if } u > \ell \cdot 2^{-\lambda_r} \\ \lfloor U/2^{k-p-1} \rfloor & \text{otherwise.} \end{cases} \tag{2.18}$$

For $(k, p, e_{\max}) = (32, 24, 127)$, some corresponding C code, for $S_r = 0$, is:

```
if(cond){
  Mr = (U - (0x40 - (Sr >> 24))) >> 7;
}else{
  Mr = (U + (Sr >> 24)) >> 7;
}
```

The above code has been given for $s_r = 0$, but notice that it also works when $s_r = 1$, since in this case, we want $m_r = \mathrm{RU}_p(\ell \cdot 2^{-\lambda_r})$. Indeed, observe now that if $s_r = 1$ then $S_r = 2^{k-1}$ and thus:

$$U + 2^{k-p-2} = U - (2^{k-p-2} + S_r \cdot 2^{-p}),$$

and

$$U + 2^{k-p-1} = U + 2^{-p},$$

that were both used to define the integer M_r in (2.17) for RoundTowardPositive, and which precisely encode $\mathrm{RU}_p(\ell \cdot 2^{-\lambda_r})$, as wanted.

Computing m_r for RoundTowardZero

For RoundTowardZero, from Table 2.1, the expected result m_r is the floating-point number such that $0 \leq \ell \cdot 2^{-\lambda_r} - m_r < 2^{1-p}$.

Clearly, whatever the sign s_r is, the RoundTowardZero algorithm is equivalent to the RoundTowardNegative for $s_r = 0$. Therefore, for $s_r \in \{0, 1\}$, the resulting integer M_r encoding the floating-point number m_r may be defined as follows:

$$M_r = \begin{cases} \lfloor (U - 2^{k-p-2})/2^{k-p-1} \rfloor & \text{if } u > \ell \cdot 2^{-\lambda_r}, \\ \lfloor U/2^{k-p-1} \rfloor & \text{otherwise.} \end{cases}$$

For $(k, p, e_{\max}) = (32, 24, 127)$, the following piece of C code implements the rounding procedure for the RoundTowardZero attribute and $s_r \in \{0, 1\}$.

```
if(cond){
  Mr = (U - 0x40) >> 7;
}else{
  Mr = U >> 7;
}
```

2.3 Packing the correctly-rounded result

This last section presents the method used for implementing the standard integer encoding of the correctly-rounded result, when $|r|$ either lies in the range $[\alpha, \Omega]$ of (sub)normal floating-point numbers or overflows.

2.3.1 Explicit formula for the standard encoding of the correctly-rounded result

How to compute the resulting integer R?

From Section 2.2.1, we know that the correctly-rounded result is defined as $\circ(r) = (-1)^{s_r} \cdot m_r \cdot 2^{e_r}$, with $m_r \in [0, 2]$ and $e_{\min} \leq e_r \leq e_{\max}$. Here, we consider the given k-bit unsigned integers S_r, M_r, and E_r encoding respectively s_r, m_r, and the biased value of e_r, such as:

$$S_r = s_r \cdot 2^{k-1}, \quad M_r = m_r \cdot 2^{p-1} \quad \text{and} \quad E_r = e_r - e_{\min} + n_r,$$

with n_r the "is normal" bit of the result (that is, $n_r = 1$ if and only if $m_r > 1$, and $n_r = 0$ otherwise). Let R be the k-bit unsigned integer encoding of r, defined using the binary interchange encoding defined in Section 1.2.2. By definition, $m_r \leq 2$ and $M_r \leq 2^p$. It follows that if $m_r < 2$, then the resulting integer R may be computed as:

$$R = S_r + E_r \cdot 2^{p-1} + (0.m_{r,1}m_{r,2}\cdots m_{r,p-1}) \cdot 2^{p-1}. \tag{2.19}$$

Otherwise, $m_r = 2$, and the resulting integer R then is simply given by:

$$R = S_r + (E_r + 1) \cdot 2^{p-1}. \tag{2.20}$$

The main drawbacks of using (2.19) and (2.20) is that it requires:

- to remove the bits of the integer part of the floating-point number m_r;
- to increment E_r by one when $m_r = 2$.

Example 2.6. *Assume* $(k, p, e_{max}) = (32, 24, 127)$. *Let* $\circ(r) = 1.414213538169860839843 75$ *be the correctly-rounded value of* $\sqrt{2.0}$ *in RoundTiesToEven. We have*

$$\circ(r) = (-1)^0 \cdot 2^0 \cdot 1.01101010000010011110011_2,$$

with $s_r = 0$, $S_r = 2^{31}$, $e_r = 0$, $n_r = 1$, $E_r = n_r - e_{min} = 127$, *and* $M_r = 1.01101010000010011110011_2 \cdot 2^{23}$. *Hence, using* (2.19) *we get the following resulting integer* R*:*

(S_r)		0 00000000 00000000000000000000000
$(E_r \cdot 2^{23})$	+	0 01111111 00000000000000000000000
$(m_r - n_r) \cdot 2^{23}$	+	0 00000000 01101010000010011110011
R	=	0 01111111 01101010000010011110011

Property 2.1. *Let* $D = E_r - n_r$. *Then*

$$R = S_r + D \cdot 2^{p-1} + M_r. \tag{2.21}$$

Proof. It suffices to show that R in (2.19) and (2.20) satisfies:

$$R = S_r + (E_r - n_r) \cdot 2^{p-1} + m_r \cdot 2^{p-1}. \tag{2.22}$$

If $m_r < 2$ then it follows from (2.19) that $R = S_r + E_r \cdot 2^{p-1} + (m_r - n_r) \cdot 2^{p-1}$, which is equivalent to (2.22).

Otherwise, since $e_r \geq e_{\min}$, if $m_r = 2$ then r cannot be a subnormal floating-point number, and thus $n_r = 1$. Then, rewriting (2.20) as $R = S_r + (E_r - 1) \cdot 2^{p-1} + 2 \cdot 2^{p-1}$ immediately gives (2.22). □

Example 2.7. *Assume* $(k, p, e_{max}) = (32, 24, 127)$. *Let* $\circ(r) = 1.414213538169860839843 75$ *the correctly-rounded value of* $\sqrt{2.0}$, *as defined in Example 2.6. We have*

$$\circ(r) = (-1)^0 \times 2^0 \cdot (1.01101010000010011110011)_2,$$

with $s_r = 0$ *and* $S_r = 2^{31}$, $e_r = 0$ *and* $D = E_r - n_r = -e_{min} = 126$, *and* $M_r = 1.01101010000010011110011_2 \cdot 2^{p-1}$. *Hence, using* (2.21) *we have the following resulting integer* R*:*

(S_r)		0 00000000 00000000000000000000000
$(D \cdot 2^{23})$	+	0 01111110 00000000000000000000000
(M_r)	+	0 00000001 01101010000010011110011
R	=	0 01111111 01101010000010011110011

Hence the computation of R using (2.21) in Property 2.1 may be done without any exponent update or bits removing. Moreover, by definition, we have $R \leq S_r + (E_r + 1) \cdot 2^{p-1}$. Since $e_r \leq e_{\max} = 1 - e_{\min}$ and since $n_r \leq 1$, we deduce that $E_r \leq 2e_{\max}$, from it follows that:

$$(E_r + 1) \cdot 2^{p-1} \leq 2^{k-1} - 2^p \quad \text{and} \quad R \leq 2^k - 2^{p-1}.$$

It follows that no carry is propagated to the sign bit when computing $(E_r + 1) \cdot 2^{p-1}$, and also that R fits in a k-bit unsigned integer. In the remainder of this thesis, our implementations will heavily rely on the computation of these three elements S_r,D, and M_r and the result will be packed according to Property 2.1.

For $(k, p, e_{\max}) = (32, 24, 127)$, Property 2.1 shows that, given S_r, D, and M_r the integers defined above, the computation of the resulting integer R may be implemented using the following C code.

```
R = (Sr + (D << 23)) + Mr;
```

We expect that the integer M_r is more expensive to compute than S_r and D. This is why we compute first `Sr + (D << 23)` before adding the correct significand.

Special case: $\ell = 2$ and $d = e_{\max}$

Consider now the case where $\ell = 2$ and $d = e_{\max}$: we deduce that $m_r = 2$ and $|r| > \Omega$. In this document, this special case concerns only the implementation of the reciprocal function, studied in Section 3.6. Since $e_{\max} = 2^{k-p} - 1$, in this case the resulting integer constructed using (2.21) is as follows:

$$R = S_r + (2^{k-1} - 2^{p-1}). \tag{2.23}$$

From Table 1.4, we may check that this integer encodes either $+\infty$ or $-\infty$.

However, when the infinitely precise result r satisfies $|r| > \Omega$, then the IEEE 754-2008 standard requires various results to be returned [IEE08, §4.3], depending on the rounding-direction attribute and the sign of the result s_r, and not only $\pm\infty$, as described below:

RoundTiesToEven. In this case the standard [IEE08, §4.3.1] requires that infinity with the correct sign be returned (Figure 2.10(a)):

$$\mathrm{RN}_p(r) = (-1)^{s_r}\infty.$$

RoundTowardPositive. In this case, from [IEE08, §4.3.2], it follows that the returned result has to be a floating-point number or an infinity no less than the exact result (Figure 2.10(b)). Hence if the result is positive ($s_r = 0$) then $+\infty$ has to be returned, while if the result is negative ($s_r = 1$), the smallest floating-point number $-\Omega$ has to be returned:

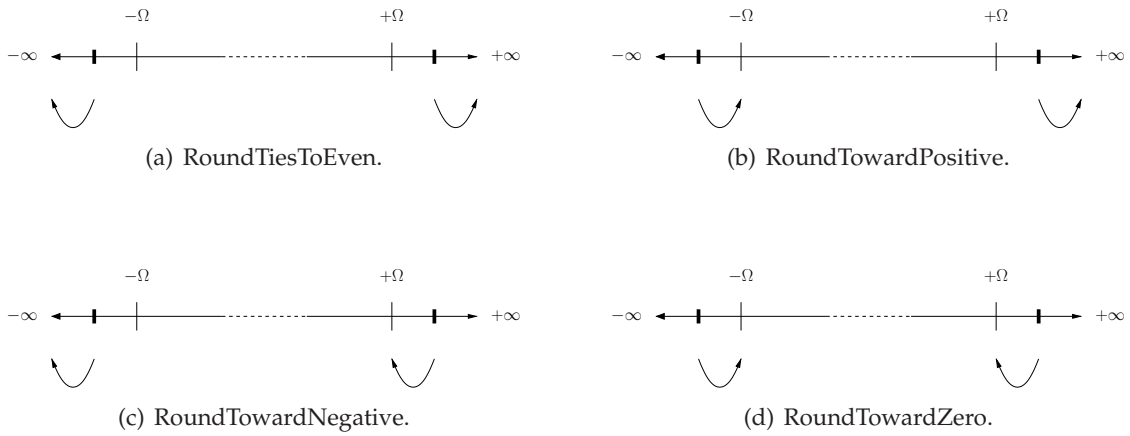
$$\mathrm{RU}_p(r) = \begin{cases} +\infty & \text{if } s_r = 0, \\ -\Omega & \text{if } s_r = 1. \end{cases}$$

RoundTowardNegative. On the contrary, in this case, from [IEE08, §4.3.2], the returned result has to be a floating-point number or an infinity no greater than the exact result (Figure 2.10(c)). Therefore if the result is negative ($s_r = 1$) then $-\infty$ has to be returned, while if the result is positive ($s_r = 0$), the largest positive floating-point number $+\Omega$ has to be returned:

$$\mathrm{RD}_p(r) = \begin{cases} +\Omega & \text{if } s_r = 0, \\ -\infty & \text{if } s_r = 1. \end{cases}$$

RoundTowardZero. In this final case, the standard requires that the result be a floating-point number no greater in magnitude than the exact result (Figure 2.10(d)). And whatever the sign of the result is, the largest floating-point $\pm\Omega$ (with the correct sign) has to be returned:

$$\mathrm{RZ}_p(r) = (-1)^{s_r}\Omega.$$



(a) RoundTiesToEven. (b) RoundTowardPositive.

(c) RoundTowardNegative. (d) RoundTowardZero.

Figure 2.10: Rounding, when overflow occurs.

Remark that if the exact result ℓ defined in (2.2) is strictly less than 2 (that is, $2 - 2^{1-p} < \ell < 2$), these cases above will be handled through the rounding procedures detailed in Section 2.2.4. However if $\ell = 2$, using these rounding procedures may lead to a result not compliant to the IEEE 754-2008 standard (see Example 2.8 below).

Example 2.8. *Assume $(k, p, e_{max}) = (32, 24, 127)$. Let x_1 and x_2 be two floating-numbers and $f(x)$ a function such as*

$$f(x_1) = \ell_1 \cdot 2^{e_{max}} \quad \text{and} \quad f(x_2) = \ell_2 \cdot 2^{e_{max}},$$

with $\ell_1 = 2$ and $2 - 2^{-24} < \ell_2 < 2$, having both an infinite binary expansion. Considering RoundTowardNegative rounding-direction attribute, in both cases, the returned result should be $+\Omega$. However it follows from the rounding procedure presented in Section 2.2.4 that $m_{r_1} = 2$ and that the returned result is $+\infty$ (instead of $+\Omega$), while on the other hand, $m_{r_2} = 2 - 2^{-23}$ and the returned result is $+\Omega$ as required by the IEEE 754-2008 standard.

For example, considering the *reciprocal* function presented in Section 3.6, the number of inputs for which these special cases ($\ell = 2$ and $e_r = e_{\max}$) occur is small [5], and they may be listed and handled separately. Section 3.6.3 presents the handling of such so-called *special implementation case* for reciprocal.

2.3.2 Overflow handling

The last case to be considered when dealing with the returned integer packing is the case where $d > e_{\max}$. In this case $e_r = d$, and $D = e_r - e_{\min}$ gives $D > 2e_{\max} - 1$. Since D is an integer, it follows that $d > e_{\max}$ if and only if $D \geq 2e_{\max}$. Once we have detected that an overflow occurs, we have to decide which result must be returned. Using the same approach as for the special case where $\ell = 2$ and $d = e_{\max}$ (presented in Section 2.3.1 above), and by definition of the rounding-direction attributes in [IEE08, §4.3] we deduce the following C codes, for $(k, p, e_{\max}) = (32, 24, 127)$.

RoundTiesToEven: $R = S_r + (2^{k-1} - 2^{p-1})$.

```
if (D >= 0xFE) return Sr + 0x7F800000;
```

[5] For the *reciprocal* function, this case occurs for two inputs: $x = \pm 2^{e_{\min}-2}$.

RoundTowardPositive: $R = S_r + (2^{k-1} - 2^{p-1} - s_r)$, with $S_r = s_r \cdot 2^{k-1}$.

```
if (D >= 0xFE) return Sr + 0x7F800000 - (Sr >> 31);
```

RoundTowardPositive: $R = S_r + (2^{k-1} - 2^{p-1} - 1 + s_r)$, with $S_r = s_r \cdot 2^{k-1}$.

```
if (D >= 0xFE) return Sr + 0x7F7FFFFF + (Sr >> 31);
```

RoundTowardZero: $R = S_r + (2^{k-1} - 2^{p-1} - 1)$.

```
if (D >= 0xFE) return Sr + 0x7F7FFFFF;
```

2.4 Example of correctly-rounded multiplication

This section presents the implementation of multiplication with correct rounding, for which the sticky bit can be computed exactly. The algorithm is well-known [MBdD+09], but we illustrate here that using the basic blocks presenting all along this chapter enables to write efficient and provably-correct code in a faster way.

This section is organized as follows. First we detail the range reduction of the multiplication in Section 2.4.1. Then, we show how to compute the correctly-rounded significand m_r in Section 2.4.2, before explaining how to implement this operation on VLIW integer processors for the *binary32* floating-point format in Section 2.4.3.

2.4.1 Range reduction

Let x and y be two (sub)normal nonzero floating-point numbers as in (1.3). This section presents the range reduction for the multiplication operation, that is, the way used to deduce the expression of the exact result r (from those of x and y) required in (2.2):

$$r = (-1)^{s_r} \cdot \ell \cdot 2^d \quad \text{with} \quad s_r \in \{0,1\}, \quad \ell \in [1,2], \quad \text{and} \quad d \in \mathbb{Z}. \tag{2.24}$$

To do so, since x, y may be subnormal numbers, we consider x and y in their normalized representation, as in (1.5):

$$x = (-1)^{s_x} \cdot m'_x \cdot 2^{e'_x} \quad \text{and} \quad y = (-1)^{s_y} \cdot m'_y \cdot 2^{e'_y},$$

with $s_x, s_y \in \{0,1\}$, $m'_x, m'_y \in [1, 2 - 2^{1-p}]$ and $e'_x, e'_y \in \{e_{\min} - p + 1, \ldots, e_{\max}\}$. It follows that the exact result r of the product $x \cdot y$ can be defined as follows:

$$r = (-1)^{s_r} \cdot (m'_x \cdot m'_y) \cdot 2^{e'_x + e'_y},$$

with $s_r = s_x$ XOR s_y. Here, by definition of m'_x and m'_y, we know that $m'_x \cdot m'_y \in [1, 4)$. Hence, let us define the value $c \in \{0,1\}$ as follows:

$$c = \begin{cases} 0 & \text{if } m'_x \cdot m'_y < 2, \\ 1 & \text{if } m'_x \cdot m'_y \geq 2. \end{cases} \tag{2.25}$$

Therefore in order to achieve the range requirements in (2.24), we can deduce the following definition for ℓ and d,

$$\ell = 2^{-c} \cdot m'_x \cdot m'_y \quad \text{and} \quad d = e'_x + e'_y + c. \tag{2.26}$$

Indeed, it follows from m'_x,$m'_y \in [1, 2-2^{1-p}]$ and from $c \in \{0,1\}$ that $\ell \in [1,2)$ and that

$$d \in \{2e_{\min}, \ldots, 2e_{\max}+1\}. \tag{2.27}$$

From (2.24) and 2.27 we deduce that the exact product may underflow or overflow, that is, we may have one of the following cases:

$$|r| < 2^{e_{\min}} \quad \text{or} \quad |r| > 2^{e_{\max}} \cdot (2-2^{1-p}).$$

This is illustrated by Example 2.9 below.

Example 2.9. *Let x_1, x_2, y_1, and y_2 four nonzero positive (sub)normal floating-point numbers defined as follows:*

$$x_1 = 2^{e_{min}-p+1}, \quad x_2 = 2^{(e_{max}+1)/2+1}, \quad y_1 = 2^{p-2}, \quad \textit{and} \quad y_2 = 2^{(e_{max}+1)/2}.$$

We finally get

$$x_1 \cdot y_1 = 2^{e_{min}-1} < 2^{e_{min}} \quad \textit{and} \quad x_2 \cdot y_2 = 2^{e_{max}+2} > 2^{e_{max}} \cdot (2-2^{1-p}).$$

From that statement, we have to distinguish the three cases presented in Section 2.2.1: $d > e_{\max}$, $e_{\min} \leq d \leq e_{\max}$, and $d < e_{\min}$. The case where $d > e_{\max}$ and the exact result r overflows is handled has presented in Section 2.3.2, and will not be discussed in more details in this section. Here we consider the two last cases:

$$d \in \{e_{\min}, \ldots, e_{\max}\} \quad \text{or} \quad d < e_{\min}.$$

As mentioned in (2.3), to handle both cases together, let us define λ_r and the exponent e_r as follows:

$$\lambda_r = \max(0, e_{\min} - d) \quad \text{and} \quad e_r = \max(e_{\min}, d),$$

with $e_r \in \{e_{\min}, \ldots, e_{\max}\}$ and $\lambda_r \in \{0, \ldots, e_{\max}-1\}$ by definition of the exponent d in (2.27). Finally, the correcly-rounded result to be returned is defined as in (2.5), that is:

$$\circ(r) = (-1)^{s_r} \times \circ(\ell \cdot 2^{-\lambda_r}) \times 2^{e_r}. \tag{2.28}$$

Remark here that the exact value ℓ cannot be equal to 2 (and $\ell \cdot 2^{-\lambda_r}$ as well, since $\lambda_r \geq 0$). Hence the case $\ell = 2$ and $d = e_{\max}$ can never occur, and no *special implementation case* has to be handled. This means that, when implementing multiplication, the returned integer computed as presented in Property 2.1 is always the expected one (for x,y two (sub)normal floating-point numbers).

2.4.2 Computation of the correctly-rounded significand

This section defines the real value ℓ, and explains how to deduce the correctly-rounded floating-point number m_r. Then, it shows how to compute the bits useful for determining the rounding bit b.

Defining the binary expansion of ℓ and deducing m_r

Let us first define the real value ℓ in (2.28), and then let us show how to deduce the correctly-rounded floating-point number m_r. By definition, the normalized significand of x and y, m'_x and m'_y, respectively, have both at most p significant bits, with $p < k$. Hence, we know that the exact

product $m'_x \cdot m'_y$ has a binary expansion on at most $2p$ bits. Let s be the exact product $m'_x \cdot m'_y$. Then

$$s = s_{-1}s_0.s_1s_2\cdots s_{2p-2} \quad \in [1,4).$$

In our context, the implementation is done using k-bit numbers. Since $k < 2p < 2k$ (which is at least the case for each of the binary floating-point formats defined in the IEEE 754-2008 standard), the binary expansion of s has to be stored with two k-bit values. Let s_{high} and s_{low} be defined as

$$s_{\text{high}} = \texttt{truncate}(s)_{k-2} \quad \text{and} \quad s = s_{\text{high}} + s_{\text{low}} \cdot 2^{2-k}. \tag{2.29}$$

This implies that

$$s_{\text{high}} = s_{-1}s_0.s_1s_2\cdots s_{k-2} \quad \text{and} \quad s_{\text{low}} = 0.s_{k-1}s_k\cdots s_{2p-2}\underbrace{000\cdots000}_{2k-2p \text{ zeros}}. \tag{2.30}$$

Moreover, by definition of the value c in (2.25), it follows that

$$c = \begin{cases} 0 & \text{if } s_{-1}s_0 = 01, \\ 1 & \text{if } s_{-1}s_0 = 1*, \end{cases}$$

that is, $c = s_{-1}$. Finally from the definition of ℓ in (2.26), it follows that

$$\ell \cdot 2^{-\lambda_r} = 0\underbrace{0.0\ldots0}_{\lambda_r \text{ zeros}}1s_{1-c}s_{2-c}\ldots s_{2p-2}, \quad \text{and} \quad \ell_i = s_{i-c}. \tag{2.31}$$

As in (2.8) let us denote by u the truncation of $\ell \cdot 2^{-\lambda_r}$ after $k-2$ fraction bits. Since $\lambda_r \geq 0$ and since $s_{-1-c} = 0$, we deduce from (2.29) and (2.31) that

$$u = \begin{cases} 0 & \text{if } \lambda_r \geq k-1, \\ \texttt{truncate}(s_{\text{high}} \cdot 2^{-(\lambda_r+c)})_{k-2} & \text{otherwise.} \end{cases}$$

Hence, we conclude that u may be defined as follows:

$$u = \texttt{truncate}(s_{\text{high}} \cdot 2^{-\min(k-1,\lambda_r+c)})_{k-2}. \tag{2.32}$$

Using the definition of the correctly-rounded floating-point number m_r in (2.10) together with the definition of truncation in (2.7), and (2.32), we finally get:

$$\begin{aligned} m_r &= \texttt{truncate}(s_{\text{high}} \cdot 2^{-\min(k-1,\lambda_r+c)})_{p-1} + b \cdot 2^{1-p} \\ &= \lfloor s_{\text{high}} \cdot 2^{-\min(k-1,\lambda_r+c)} \cdot 2^{p-1} \rfloor / 2^{p-1} + b \cdot 2^{1-p} \end{aligned} \tag{2.33}$$

In this case, we can compute m_r directly from the value s_{high}, and more particularly without computing first the value u, which will not be the case for the functions studied in Chapters 3 and 4.

We know how to compute s_{high}. It remains now to see how to determine the rounding bit b, and more particularly the bits useful for computing it.

Determining the bits useful for computing the rounding bit b

Let us deduce now, from the binary expansion of $\ell \cdot 2^{-\lambda_r}$, the three bits useful for computing the rounding bit b, as given in Section 2.2.3. These three bits are $\ell_{p-1-\lambda_r}$, the guard bit $g = \ell_{p-\lambda_r}$, and the sticky bit s. From (2.31), it follows that:

$$\ell_{p-1-\lambda_r} = s_{p-1-c-\lambda_r}, \quad g = s_{p-c-\lambda_r}, \quad \text{and} \quad s = s_{p+1-c-\lambda_r} \vee \cdots \vee s_{2p-2}. \tag{2.34}$$

Remark that depending on the rounding-direction attribute considered, some of these bits may be not used. But for the sake of generality of this section, we present now the way used to decide each of them.

- We know from (2.31) that the bit $\ell_{p-1-\lambda_r} = s_{p-1-c-\lambda_r}$, and since $s_{-1-c} = 0$, then $\ell_{p-1-\lambda_r}$ is defined as follows:
$$\ell_{p-1-\lambda_r} = \begin{cases} 0 & \text{if } \lambda_r \geq p, \\ s_{p-1-c-\lambda_r} & \text{otherwise.} \end{cases}$$
It follows that the bit $\ell_{p-1-\lambda_r}$ can always be deduced from the bit string of s_{high}. Let S_{high} be the k-bit unsigned integer encoding s_{high} such as:
$$S_{\text{high}} = s_{\text{high}} \cdot 2^{k-2}. \tag{2.35}$$
Hence, the bit $\ell_{p-1-\lambda_r}$ may be extracted from the bit string of S_{high} by shifting right S_{high} by $k-1-p+c+\min(p,\lambda_r)$ bits, and by taking the bitwise AND between the resulting integer and 1:
$$\ell_{p-1-\lambda_r} = \big(S_{\text{high}} \gg (k-1-p+c+\min(p,\lambda_r))\big) \quad \text{AND} \quad 1.$$
However, if $\lambda_r \geq p$ and $c = 1$, then S_{high} is shifted right by k bits, then the shift operation is not well-defined in the sense of the C standard [Int99], that requires the shift to be of strictly less than k bits. In this case, we may have an *undefined behavior*. Hence, to solve this problem, we first shift S_{high} right by $k-1-p$ bits and then by $c+\min(p,\lambda_r)$. We finally get:
$$\ell_{p-1-\lambda_r} = \big(S_{\text{high}} \gg (k-1-p)\big) \gg (c+\min(p,\lambda_r)) \quad \text{AND} \quad 1, \tag{2.36}$$
with $0 \leq k-1-p, c+\min(p,\lambda_r) < k$. We observe also that this solution enables to expose sligthly more instruction-level parallelism than the previous one.

- From (2.31) and (2.34), we know that $g = s_{p-c-\lambda_r}$. More precisely, we have:
$$g = \begin{cases} 0 & \text{if } \lambda_r \geq p+1 \\ s_{p-c-\lambda_r} & \text{otherwise.} \end{cases}$$
Actually, the guard bit g can be extracted from the bit string of S_{high} in (2.35) in the same way as presented above for $\ell_{p-1-\lambda_r}$. It follows that g is defined as:
$$g = \big(S_{\text{high}} \gg (k-2-p)\big) \gg (c+\min(p+1,\lambda_r)) \quad \text{AND} \quad 1, \tag{2.37}$$
with $0 \leq k-2-p$ and $c+\min(p+1,\lambda_r) < k$.

- Finally, from (2.9), (2.31), and (2.34), it follows that the sticky bit s is defined as a "logical or" of the bits s_i for $i \in \{p-c-\lambda_r+1, \ldots, 2p-2\}$. Since $c \geq 0$ and $\lambda_r \geq 0$, we deduce from (2.30) that the sticky bit s equals 0 if and only if
$$s_{p-c-\lambda_r+1} \cdots s_{k-2} = 0 \quad \text{and} \quad s_{\text{low}} = 0.$$
Let S_{low} be the k-bit unsigned integer encoding s_{low} such as:
$$S_{\text{low}} = s_{\text{low}} \cdot 2^{k}. \tag{2.38}$$
It is obvious that $s_{\text{low}} = 0$ if and only if $S_{\text{low}} = 0$. It remains now to decide if $s_{p-c-\lambda_r+1} \cdots s_{k-2}$ equals 0. Notice first that $\lambda_r \geq p+2-c$ implies $s_{p-c-\lambda_r+1} \cdots s_{k-2} = s_{\text{high}}$. It follows that the bits $s_{p-c-\lambda_r+1}, \cdots, s_{k-2}$ can be extracted from the bit string of S_{high} by removing its $p+2-c-\lambda_r$ leading bits, which may be done by shifting S_{high} left by $\max(0, p+2-c-\lambda_r)$ bits, since $p+2-c-\lambda_r$ may be less than 0. Finally, the sticky bit is defined as follows:
$$s = \big((S_{\text{high}} \ll \max(0, p+2-c-\lambda_r)) \neq 0\big) \vee \big(S_{\text{low}} \neq 0\big). \tag{2.39}$$

In this section, we have defined all the elements useful for the implementation of correctly-rounded multiplication. Let us now explain how to implement this operator on the ST231 processor, for the *binary32* floating-point format.

$x \times y$		y			
		± 0	(sub)normal	$\pm\infty$	NaN
x	± 0	± 0	± 0	qNaN	qNaN
	(sub)normal	± 0	$\circ(x \times y)$	$\pm\infty$	qNaN
	$\pm\infty$	qNaN	$\pm\infty$	$\pm\infty$	qNaN
	NaN	qNaN	qNaN	qNaN	qNaN

Table 2.2: Specification of the multiplication operation $x \times y$.

2.4.3 Implementation of multiplication in the *binary32* format

This section presents in Listing 2.2 a complete code implementing multiplication for the *binary32* floating-point format. Let x and y be two floating-point data, encoded in the k-bit unsigned integers X and Y, respectively, according to the interchange encoding defined in Section 1.2.2. Here, we do not discuss how to handle special inputs. Indeed, the manner used to decide whether x or y is a special input is done as presented for division in Section 4.3, Chapter 4. And the output result is determined according to the specification of the multiplication in Table 2.2 above. In Listing 2.2, this is done between lines 15 and 24.

Therefore, from now on we consider that x and y are two (sub)normal floating-point numbers. In Listing 2.2, the input unpacking in done in lines 9, 12, and 27 to 31, as presented in Listing 2.1. Here we detail the implementation for the RoundTiesToEven rounding-direction attribute: handling of overflow (line 47, Listing 2.2), and implementation of the rounding bit b (line 49, Listing 2.2) are done as presented in Section 2.3.2 and Section 2.2.3, respectively, for RoundTiesToEven. (From Section 2.3.2 and Section 2.2.3, the correctly-rounded implementation for other rounding-direction attribute may be derivated easily, by replacing those two lines.) It remains finally to explain how to implement the computation of sign S_r, the exponent D, and the scaling λ_r, and then how to implement the computation of the resulting integer R.

Computing the result sign S_r, the exponent D, and the scaling λ_r

The computation of the sign S_r is done by extracting first the sign of the input X and Y, and by taking the XOR between both of them. Using Listing 2.1, it follows that the sign S_r may be computed, for $(k, p, e_{\max}) = (32, 24, 127)$, as presented in lines 9 and 10 in Listing 2.2.

Moreover, let E_r be the w-bit unsigned integer encoding the biased value of e_r such as $E_r = e_r - e_{\min} + n_r$. From (2.21), we know that the goal is to compute $D = E_r - n_r$ instead of E_r, that is, $D = e_r - e_{\min}$. From (2.4), we know also that $e_r = \max(e_{\min}, d)$, and from the definition of d in (2.26), we conclude that $D = \max(0, d - e_{\min})$, that is:

$$D = \max(0, e'_x + e'_y + c - e_{\min}). \tag{2.40}$$

Let E_x and E_y be the w-bit unsigned integers encoding e_x and e_y, respectively. It follows from Section 1.2.2 that

$$E_x = e_x - e_{\min} + n_x \quad \text{and} \quad E_y = e_y - e_{\min} + n_y,$$

with n_x and n_y the "is normal bit" of the floating-point number x and y, respectively. Using (1.5) and (2.1), together with (2.40), we finally get:

$$D = \max\big(0, E_x + E_y - n_x - n_y - MX - MY + c + (2w + e_{\min})\big). \tag{2.41}$$

Finally, let us denote by L_r the k-bit unsigned integer encoding of λ_r. From (2.3), we know that $\lambda_r = \max(0, e_{\min} - d)$. Using the same approach as for the implementation of D, it follows that:

$$L_r = \max\big(0, -(E_x + E_y - n_x - n_y - MX - MY + c + (2w + e_{\min}))\big). \tag{2.42}$$

The implementation of D and L_r, for $(k, p, e_{\max}) = (32, 24, 127)$, is done in line 37 in Listing 2.2. (Remark that c may easily extracted by shifting right S_{high} by $k-1$ bits, as in line 34.)

Computing the resulting integer R

It remains to compute the k-bit resulting integer R, as in (2.21): $R = S_r + D \cdot 2^{p-1} + M_r$, with M_r the k-bit unsigned integer encoding of the floating-point number m_r defined in (2.33), as in (2.16):

$$M_r = m_r \cdot 2^{p-1} \quad \text{and} \quad \lfloor S_{\text{high}} \cdot 2^{1+p-k} \cdot 2^{-\min(k-1,\lambda_r+c)} \rfloor + b.$$

Here, the rounding bit b is expected to be more expensive to be computed than $\lfloor S_{\text{high}} \cdot 2^{1+p-k} \cdot 2^{-\min(k-1,\lambda_r+c)} \rfloor$. Hence, we will parenthesize the computation of R as follows:

$$R = \Big((S_r + D \cdot 2^{p-1}) + \lfloor S_{\text{high}} \cdot 2^{1+p-k} \cdot 2^{-\min(k-1,\lambda_r+c)} \rfloor \Big) + b,$$

as implemented in line 51 in Listing 2.2, for $(k, p, e_{\max}) = (32, 24, 127)$. Recall that M_{px} and M_{py} are the k-bit unsigned integers encoding of the significands m'_x and m'_y, respectively, as in Section 2.1.3:

$$M_{px} = m'_x \cdot 2^{k-1} \quad \text{and} \quad M_{py} = m'_y \cdot 2^{k-1}. \tag{2.43}$$

Recall also that S_{high} and S_{low} are the k-bit unsigned integers encoding of s_{high} and s_{low}, respectively, as in (2.35) and (2.38). From the definition of the function `mul` and the operator `*` defined in Table 1.1, as:

$$\texttt{mul}(X, Y) = \lfloor (X \cdot Y)/2^k \rfloor \quad \text{and} \quad X \texttt{*} Y = (X \cdot Y) \bmod 2^k,$$

it follows together with (2.29) and (2.43) that:

$$S_{\text{high}} = \texttt{mul}(M_{px}, M_{py}) \quad \text{and} \quad S_{\text{low}} = M_{px} * M_{py},$$

as shown in line 33 (Listing 2.2). Finally, in integer arithmetic, $\lfloor S_{\text{high}} \cdot 2^{1+p-k} \cdot 2^{-\min(k-1\lambda_r+c)} \rfloor$ can be simply implemented using two shifting operations, as in line 45:

$$\big(S_{\text{high}} \gg (k-1-p)\big) \gg \min(k-1, \lambda_r + c).$$

It remains to explain the implementation of the rounding bit b in line 49. Let s_1 and s_2 be such as $s = s_1 \vee s_2$ with:

$$s_1 = \big(S_{\text{high}} \ll \max(0, p+2-c-\lambda_r)\big) \neq 0 \quad \text{and} \quad s_2 = S_{\text{high}} \neq 0,$$

so that $b = g \wedge \big(\ell_{p-1-\lambda_r} \vee (s_1 \vee s_2)\big)$. Since s_1 is expected to be more expensive to be computed than $\ell_{p-1-\lambda_r}$ and s_2, we define b, for RoundTiesToEven, as follows:

$$b = g \wedge \big((\ell_{p-1-\lambda_r} \vee s_1) \vee s_2\big). \tag{2.44}$$

Finally, from (2.36), (2.37), and (2.39), the implementation of the bits useful for computing b is done from lines 40 to 43 (Listing 2.2), where `lsb` encodes $\ell_{p-1-\lambda_r}$. Remark finally that, since $\big((\ell_{p-1-\lambda_r} \vee s_1) \vee s_2\big) \in \{0, 1\}$ in (2.44), the bitwise AND in (2.37) is not necessary, and is actually not used in line 40 of Listing 2.2.

```
1  uint32_t flip_binary32_mul(uint32_t X , uint32_t Y)
2  {
3    uint32_t absX, absY, Min, Max, Inf;
4    uint32_t Sx, Lx, nx, Ex, MX, mpX, Sy, Ly, ny, Ey, MY, mpY;
5    uint32_t R = 0, Sr, D, Lr, Mr_minus_b, Shigh, Slow, c;
6    uint32_t b, g, lsb, s1, s2;
7    int32_t tmp;
8
9    Sx = X & 0x80000000;             Sy = Y & 0x80000000;
10   Sr = Sx ^ Sy;
11
12   absX = X & 0x7FFFFFFF;           absY = Y & 0x7FFFFFFF;
13
14   // Special value handling // see Section 4.3 for more details
15   if (maxu(absX-1, absY-1) >= 0x7F7FFFFF) {
16     Min = minu(absX, absY); Max = maxu(absX, absY); Inf = Sr | 0x7F800000;
17
18     if (Max > 0x7F800000 || (Min == 0 && Max == 0x7F800000))
19       return (Inf | 0x00400000) | Max;     // qNaN with payload equal to
20                                            // the last 22 bits of X or Y
21     if (Max != 0x7F800000)
22       return Sr;
23
24     return Inf;
25   }else{
26     // Floating-point number handling: X,Y != {-0,+0,-Inf,+Inf,NaN}
27     Ex = absX >> 23;                Ey = absY >> 23;
28     nx = absX >= 0x00800000;        ny = absY >= 0x00800000;
29     Lx = nlz(absX);                 Ly = nlz(absY);
30     MX = maxu(Lx , 8);              MY = maxu(Ly , 8);
31     mpX = (X << MX) | 0x80000000;   mpY = (Y << MY) | 0x80000000;
32
33     Shigh = mul(mpX , mpY);         Slow = mpX * mpY;
34     c = Shigh >> 31;
35
36     tmp = Ex + Ey - nx - ny - MX - MY + c - 110;
37     D  = max( 0 , tmp );            Lr = max( 0 , 0x0 - tmp );
38
39     // Bits useful for rounding
40     g   = (Shigh >> 6) >> (c + min(Lr , 25));
41     lsb = (Shigh >> 7) >> (c + min(Lr , 24))&0x1;
42     s1  = (Shigh << max(0 , 26 - c - Lr)) != 0;
43     s2  = Slow != 0;
44
45     Mr_minus_b = (Shigh >> 7) >> min(31 , Lr + c);
46
47     if (D >= 0xFE) return Sr | 0x7F800000;
48
49     b = g & ((lsb | s2) | s1);
50
51     R = ((Sr | (D << 23)) + Mr_minus_b) + b;
52   }
53   return R;
54 }
```

Listing 2.2: Correctly-rounded multiplication for the *binary32* format and RoundTiesToNearest rounding-direction attribute.

CHAPTER 3

A uniform approach for correctly-rounded roots and their reciprocals

This chapter presents a uniform approach for the implementation of roots and their reciprocals, based on the evaluation of a particular bivariate polynomial*. It turns out that this approach is more efficient on VLIW integer processors (like the ST231) than the classical ones (*iterative methods*,* multiplicative methods*, or* univariate polynomial-based methods*). This approach is presented here in a fully parametrized way, with detailed analyses. Its efficiency is mainly achieved thanks to the optimized and certified polynomial evaluation pogram generated using CGPE (Part II). Finally two detailed examples of such an implementation for the* binary32 *format are given: the square root and the reciprocal. Notice that this approach has already been used for implementing several other functions of FLIP, with an average speedup of a factor of about 1.85 compared to those of FLIP 0.3.*

This chapter presents a uniform approach for implementing, for a given $n \in \mathbb{Z} \setminus \{0, 1\}$, the function $x^{1/n}$, with correct rounding and optimized for VLIW integer processors. Recall that our work aims at the design of some basic mathematical functions optimized for the ST231, a 4-issue VLIW integer processor of the ST200 family of cores of STMicroelectronics. ST200 processors are embedded media processors highly used in the audio and video domains, and in particular designed to implement advanced audio and video codecs in consumers devices (set-top boxes for HP-IPTV, cell phones, wireless terminals, and PDAs). In these domains, it can be useful to have an efficient support for functions like square root $x^{1/2}$, cube root $x^{1/3}$, ... and their reciprocals.

Many different algorithms are known for the implementation in hardware and software of such functions (see [PPB03], and, for square root for example, the survey [MM90] or the reference books [EL04], [Mar00], or [CHT02]). *Iterative* methods (restoring, nonrestoring, SRT,...) are based on an iterative process that produces one or a few bits of the result per iteration (see [EL94] or [PB02] for square root, and [PBLM08] for cube root, or [MBCP07] for an extension to general nth roots, for example). Unfortunately, these methods, that have linear convergence, are ill-suited in our context, since they are highly sequential. Indeed their implementation uses most probably only 1 issue out of the 4 available on the target architecture, and thus may be slow. *Multiplicative* methods [EL04, §7] (Newton-Raphson or Goldschmidt methods [EIM+00]) converge quadratically by refining a first approximation of the function, that may be obtained in various ways. For example, reciprocal square root is implemented via Newton-Raphson algorithm in [FHL+07],

[Zim08], [BZ09], or Newton-like iterations in [SW99]. On HP/Intel's Itanium processor, the square root is implemented using a Newton-Raphson algorithm that refines a first approximation obtained by calling a specific hardware instruction `frsqrta` (on the IA-64 architecture) that approximates the reciprocal square root to around 8 bits [Mar00, §9.1.1,§9.3.1], [CHN99], [GHH+01], [CHT02, p.238], [Mar04]. However this approach requires that this kind of instruction be available, which is not the case on most architectures. On IBM's RS/6000 [Mar90] or Power3 [AGS99], the first approximation is obtained by look-up table and refined by Newton-Raphson iteration, while on AMD-K7™ it is refined using Goldschmidt-like iteration [Obe99]. It may also be obtained by evaluating small-degree polynomial approximants: for example in the previous version of square root and reciprocal square root in FLIP 0.3 [Rai06, §11, §12], the first approximation was computed via the evaluation of degree-3 polynomial approximants and refined using Goldschmidt methods; a similar example, but with degree-2 polynomials and in hardware, can be found in [PB02]. The last methods that may be used for implementing such functions are *polynomial-based* methods. For example, in [JKMR07] we have implemented the square root via the evaluation of several small-degree univariate polynomials, while in [AGS99] it is implemented through the combination of look-up tables and univariate Chebychev's polynomial evaluation.

The uniform approach we propose here relies on the efficient evaluation of a *single bivariate* polynomial. The interest of this fully parametrized approach is that it can be easily integrated into an automatic tool to generate code for the implementation of nth roots, and then derive quickly several implementations. It has already been used to implement several functions: square root (see [JKMR08] and Section 3.5), division (see [JKM+09] and Chapter 4), reciprocal (see Section 3.6), or reciprocal square root (see [JR09a]), and enables to achieve implementations about 1.85 times faster than the implementations of FLIP 0.3.

This chapter is organized as follows: Section 3.1 presents a fully parameterized approach for implementing nth roots and their reciprocals; Section 3.2 discusses how to approximate the function by a particular *single bivariate* polynomial; Section 3.3 details how to compute the sign and the exponent of the result; and Section 3.4 explains how to handle special input in all cases (n positive or negative). Finally in Sections 3.5 and 3.6, we present two detailed implementation examples for, respectively, the square root function and the reciprocal function.

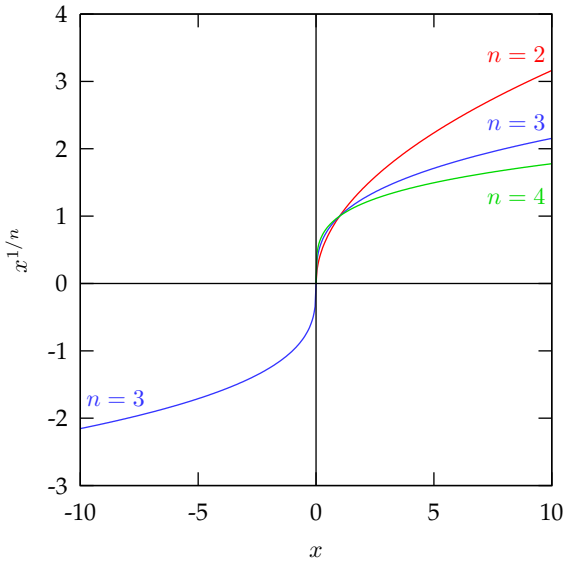
3.1 General properties of the real function $x \mapsto x^{1/n}$

Here and hereafter we shall consider the root functions and their reciprocals as special cases of the real function $x \mapsto x^{1/n}$ over $\mathbb{R}$, for given values $n \in \mathbb{Z}^*$, recommended in the IEEE 754-2008 standard [IEE08, §9.2]. More precisely, here let us define $x^{1/n}$ over $\mathbb{R}$ as follows:

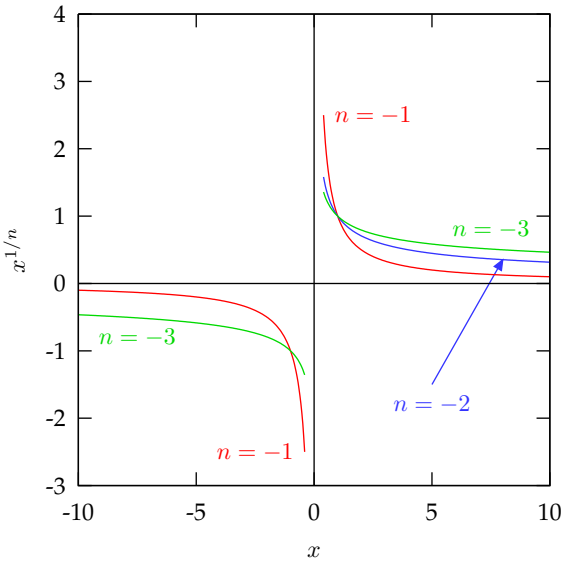
- If $x \in \mathbb{R}_{>0}$ then $x^{1/n}$ is defined as the unique $y \in \mathbb{R}_{>0}$ such that $y^n = x$;
- If $x \in \mathbb{R}_{<0}$ then $x^{1/n} = -(-x)^{1/n}$ if n is odd, and is undefined if n is even;
- If $x = 0$ then $x^{1/n} = 0$ if $n > 0$, and is undefined if $n < 0$.

Remark that, for example, the MPFR library [FHL+07] provides, in addition to the specific implementation of the square root, cube root, and reciprocal square root [Zim08], an implementation of the function `root`(x, n). On the contrary, here, we present a general framework for implementing the function $x^{1/n}$ for given values n (square root $x^{1/2}$, reciprocal square root $x^{-1/2}$, cube root $x^{1/3}$, ...). Figure 3.1 shows the graphs obtained by applying this definition of $x^{1/n}$ to various values n: Figure 3.1(a) shows the graphs where $n > 0$ for square root ($n = 2$), cube root ($n = 3$), and fourth root ($n = 4$), while Figure 3.1(b) displays those where $n < 0$ for reciprocal ($n = -1$), reciprocal square root ($n = -2$), and reciprocal cube root ($n = -3$).

In this chapter, we will not consider the case $n = 1$, which obviously yields the identity function. Without loss of generality, we consider that $|n| \geq 2$ or $n = -1$.



(a) Case where $n > 0$: graphs of square root ($n = 2$), cube root ($n = 3$), and fourth root ($n = 4$).



(b) Case where $n < 0$: graphs of reciprocal ($n = -1$), reciprocal square root ($n = -2$), and reciprocal cube root ($n = -3$).

Figure 3.1: Graph of $x^{1/n}$ for various values of n.

Let x be a floating-point datum defined as in Section 1.2.1 and [IEE08, §3.3]. It follows from the above definition of the function $x^{1/n}$ that we can restrict the study to positive input, while negative input will be handled as follows:

- if n is even, then x negative is considered as special input, just like ± 0, $\pm\infty$, or NaN;
- if n is odd, then x negative is considered as a positive input, with the correct sign adjoined to the result.

From that statement, in this section, we consider finally that x is a nonzero positive (sub)normal floating-point number:

$$x = m'_x \cdot 2^{e'_x} \quad \text{with} \quad m'_x \in [1, 2 - 2^{1-p}] \quad \text{and} \quad e'_x \in \{e_{\min} - p + 1, \ldots, e_{\max}\}.$$

The goal is thus to express the exact result $r = x^{1/n}$ as required in Section 2.2.1 to ensure correct rounding, that is, recalling (2.2), as

$$r = \ell \cdot 2^d, \quad \text{with} \quad \ell \in \mathbb{R} \cap [1, 2] \quad \text{and} \quad d \in \mathbb{Z}. \tag{3.1}$$

Here we do not specify any range for the exponent d: according to its value, either the exact result r overflows or lies in the range of (sub)normal floating-point numbers; and all these cases are handled as presented in Section 2.2.1. Recall that to handle them, we define λ_r and e_r as in (2.3):

$$\lambda_r = \max(0, e_{\min} - d) \quad \text{and} \quad e_r = \max(e_{\min}, d). \tag{3.2}$$

Finally the exact result r and its correctly-rounded value $\circ(r)$ are as follows:

$$r = (\ell \cdot 2^{-\lambda_r}) \cdot 2^{e_r} \quad \text{and} \quad \circ(r) = m_r \cdot 2^{e_r}, \quad \text{with} \quad m_r = \circ(\ell \cdot 2^{-\lambda_r}). \tag{3.3}$$

In the remainder of this section, we give some properties of the function $x^{1/n}$ in binary floating-point arithmetic, before detailing the range reduction, and thus giving explicit formulas for the values ℓ and d in (3.1). Finally we expose some properties that are useful for correct rounding, especially to deduce whether r can be halfway between two floating-point numbers, or exactly a floating-point number.

3.1.1 Basic properties of $x^{1/n}$ in binary floating-point arithmetic

Recall that x is a nonzero positive (sub)normal floating-point number defined in (1.5), and that $r > 0$ is the exact result of $x^{1/n}$. We have

$$x = m'_x \cdot 2^{e'_x} \quad \text{and} \quad r = x^{1/n},$$

with $m'_x \in [1, 2 - 2^{1-p}]$, $e'_x \in \{e_{\min} - p + 1, \ldots, e_{\max}\}$, and $n \in \mathbb{Z} \setminus \{0, 1\}$. The following properties deal with the range in which the exact result $r = x^{1/n}$ lies, depending on the value of n. These properties will allow us to determine if the exact result r denormalizes or not.

Neither underflow nor overflow occur when $|n| \geq 2$

We show in Property 3.1 below that, when $|n| \geq 2$, the exact result $r = x^{1/n}$ lies always in the range of normal floating-point numbers. We can deduce that the exact result cannot denormalize, and also never underflows nor overflows. This will simplify the implementation of $x^{1/n}$, since we will not have to detect and handle the case where $r < 2^{e_{\min}}$ or $r \geq 2^{e_{\max}+1}$ to decide the result that shall be returned.

Property 3.1. *For x a nonzero positive (sub)normal floating-point number defined in (1.5) in Section 1.2.1 and $|n| \geq 2$, the real value $r = x^{1/n}$ lies in the range of normal floating-point numbers, that is,*

$$x^{1/n} \in [2^{e_{min}}, \Omega].$$

Proof. By definition, we have $2^{1-p+e_{\min}} \leq x < 2^{1+e_{\max}}$. Assume first that $n \leq -2$. The function $x \mapsto x^{1/n}$ being monotonically decreasing, we deduce that $2^{(1+e_{\max})/n} < x^{1/n} \leq 2^{(1-p+e_{\min})/n}$. To get the lower bound $2^{e_{\min}} \leq x^{1/n}$, let us show that $e_{\min} \leq (1+e_{\max})/n$. Since $n < 0$ and $e_{\min} = 1-e_{\max}$, this means $n \cdot (1-e_{\max}) \geq 1+e_{\max}$, that is, $|n| \cdot (e_{\max}-1) \geq 1+e_{\max}$. This last inequality is true because $|n| \geq 2$ and because $e_{\max} \geq 2$ is odd, thus $e_{\max} \geq 3$. On the other hand, since $e_{\max} = 1 - e_{\min}$ and $p \leq 1 - e_{\min}$, we have $1 - p + e_{\min} \geq -2e_{\max}$, and using $n \leq -2$, we deduce that $2^{(1-p+e_{\min})/n} \leq 2^{e_{\max}}$. From $1 \leq 2 - 2^{1-p}$, we conclude the upper bound $x^{1/n} \leq (2 - 2^{1-p}) \cdot 2^{e_{\max}}$.

Assume now that $n \geq 2$. The function $x \mapsto x^{1/n}$ being monotonically increasing, we deduce that $2^{(1-p+e_{\min})/n} \leq x^{1/n} < 2^{(1+e_{\max})/n}$. Since $p \leq 1 - e_{\min}$ and $n \geq 2$, we deduce the lower bound $2^{e_{\min}} \leq x^{1/n}$. Moreover from $e_{\max} \geq 1$ and $n \geq 2$ we have $2^{(1+e_{\max})/n} < 2^{e_{\max}}$, and $p \geq 1$ implies $1 \leq 2 - 2^{1-p}$. Thus we deduce the upper bound $x^{1/n} \leq (2 - 2^{1-p}) \cdot 2^{e_{\max}}$, that concludes the proof. □

Recall from (3.1) that $r = \ell \cdot 2^d$ with $\ell \geq 1$ and, from (3.2), that $\lambda_r = \max(0, e_{\min} - d)$. Hence Property 3.1 implies:

$$d \geq e_{\min} \quad \text{and} \quad \lambda_r = 0, \quad \text{if} \quad |n| \geq 2. \tag{3.4}$$

From Property 3.1, and by definition of rounding-direction attributes in Section 2.2.1, we deduce that the correctly-rounded value $\circ(r)$ lies also in the range of normal floating-point numbers. That means especially that no overflow may occur while rounding. From that statement, it follows from (3.2) and (3.3) that r and $\circ(r)$ are defined as follows:

$$r = \ell \cdot 2^{e_r} \quad \text{and} \quad \circ(r) = m_r \cdot 2^{e_r}, \quad \text{with} \quad m_r = \circ(\ell) \quad \text{and} \quad e_r \in \{e_{\min}, \ldots, e_{\max}\}. \tag{3.5}$$

When $n = -1$, the exact result may underflow or overflow

For $n = -1$, the function $x^{1/n}$ obviously corresponds to the reciprocal function. In this case, the exact result $r = 1/x$, with x being a nonzero positive (sub)normal floating-point number may underflow (that is, $|r| < 2^{e_{\min}}$) or overflow (that is, $|r| > (2 - 2^{1-p}) \cdot 2^{e_{\max}}$).

Example 3.1. *Assume $p \geq 3$. Let $x = 2^{e_{max}}$ and $y = 2^{e_{min}-p+1}$. Since $e_{max} = 1 - e_{min}$, we have*

$$1/x = 2^{e_{min}-1} < 2^{e_{min}} \quad \text{and} \quad 1/y = 2^{e_{max}-2+p} > 2^{e_{max}} \cdot (2 - 2^{1-p}).$$

The characterization of underflow and overflow is presented in Section 3.6.3 discussing the implementation of the reciprocal. Here we focus on the representation of the exact result r and its correctly-rounded value $\circ(r)$. In this case, if an underflow occurs, since $\ell \leq 1$, then we know that the exponent d is strictly less than $e_{\min}$, and that $\lambda_r \neq 0$ in (3.2).

Property 3.2. *For x a nonzero positive (sub)normal floating-point number defined in (1.5) in Section 1.2.1 and $n = -1$, the real value x^{-1} lies in the range*

$$2^{e_{min}-2} < x^{-1} \leq 2^{e_{max}-2+p}. \tag{3.6}$$

Proof. By definition, $2^{1-p+e_{\min}} \leq x < 2^{1+e_{\max}}$. By definition $n = -1$. The function $x \mapsto x^{-1}$ being monotonically decreasing, we deduce $2^{-(1+e_{\max})} < x^{-1} \leq 2^{-(1-p+e_{\min})}$. Since $e_{\min} = 1 - e_{\max}$, we have finally $2^{e_{\min}-2} < x^{-1} \leq 2^{e_{\max}-2+p}$. □

More precisely, we deduce from Property 3.2 above that, when $p \geq 3$ (which is the case for the binary floating-point formats in the standard [IEE08]), the exact result $r = 1/x$ may overflow. Otherwise, when $p \leq 2$, the exact result cannot overflow.

Another interesting consequence of Property 3.2 is that λ_r in (3.2) lies in a very small range. More precisely, combining (3.6) with the fact that $\ell \in [1, 2]$ gives $2^{e_{\min}-2} < 2^{d+1}$ and $2^d \leq 2^{e_{\max}-2+p}$. Therefore,

$$e_{\min} - 2 \leq d \leq e_{\max} - 2 + p. \tag{3.7}$$

Since $\lambda_r = \max(0, e_{\min} - d)$, it follows that

$$\lambda_r \in \{0, 1, 2\}. \tag{3.8}$$

And when $d \leq e_{\max}$, the correctly-rounded result $\circ(r)$ of r that we shall return, is defined as

$$\circ(r) = m_r \cdot 2^{e_r}, \quad \text{with} \quad m_r = \circ(\ell \cdot 2^{-\lambda_r}) \quad \text{and} \quad e_r \in \{e_{\min}, \ldots, e_{\max}\}.$$

As we will see in Section 3.6 when discussing the implementation of the reciprocal function, this property will make the implementation more complicated since we will need to detect overflow, and compute the value λ_r in order to deduce the correctly-rounded result $\circ(r)$.

3.1.2 Range reduction of $x^{1/n}$

The goal of the section is now to deduce from the expression of the input $x = m'_x \cdot 2^{e'_x}$ an expression for the exact result $r = x^{1/n}$, and more particularly for ℓ and d in (3.1). We know that the exact result is $r = x^{1/n}$, with $x^{1/n} = (m'_x)^{1/n} \cdot 2^{e'_x/n}$ by definition of x. Remark that, by definition of $m'_x \in [1, 2 - 2^{1-p}]$, the exact value $(m'_x)^{1/n}$ satisfies:

$$(m'_x)^{1/n} \in (1/2, 1] \quad \text{if } n \leq -1, \quad \text{and} \quad (m'_x)^{1/n} \in [1, 2) \quad \text{if } n \geq 2.$$

Recall that we have to ensure that the value ℓ in (3.1) is a real value lying in the range $[1, 2]$. Hence, we will distinguish now two cases. To do so, let us define μ as follows:

$$\mu = \begin{cases} 0, & \text{if } n \geq 0, \\ 1, & \text{if } n < 0. \end{cases} \tag{3.9}$$

Before detailing the range reduction of the function $x^{1/n}$, let us also recall the useful *mod* notation in Definition 3.1 below.

Definition 3.1 (modulo). *Let $a, b \in \mathbb{Z}$, with $b \neq 0$. We define $a \bmod b$ as the remainder of the quotient of a divided by b [GKP94, §3.4]. In other words,*

$$a \bmod b = a - b \cdot \lfloor a/b \rfloor,$$

where $\lfloor \cdot \rfloor$ denotes the usual floor function.

Let us now detail the range reduction precisely. Let x be a nonzero positive (sub)normal floating-point number defined as $x = m'_x \cdot 2^{e'_x}$ with $m'_x \in [1, 2 - 2^{1-p}]$. Using (3.9), we define the value c as follows:

$$c = (e'_x + \mu) \bmod n. \tag{3.10}$$

Therefore, using Definition 3.1 above, we can write the exact result $r = x^{1/n}$ as:

$$r = \ell \cdot 2^d, \quad \text{with} \quad \ell = (2^{c-\mu} \cdot m'_x)^{1/n} \quad \text{and} \quad d = \lfloor (e'_x + \mu)/n \rfloor. \tag{3.11}$$

More particularly, we write the exact value ℓ in (3.11) as follows:

$$\ell = s \cdot (2^{-\mu} \cdot m'_x)^{1/n} \quad \text{with} \quad s = 2^{c/n}. \tag{3.12}$$

It follows from Lemma 3.1 below that the real number s satisfies $s \in [1, 2)$, and, more precisely, that it can only take the following n values:

$$s \in \left\{ 2^{i/|n|} \right\}_{i=0,\ldots,|n|-1}. \tag{3.13}$$

Lemma 3.1. *Let $a, b \in \mathbb{Z}$ and $b \neq 0$. Define $c = a \bmod b$. Then*

$$|c| \in \{0, \ldots, |b| - 1\} \quad \textit{and} \quad c/b \in \{0, \ldots, (|b| - 1)/|b|\}.$$

Proof. From Definition 3.1, we know that $c = a - b \cdot \lfloor a/b \rfloor$. Assume first that $b > 0$. By definition of the *floor* function, $\lfloor a/b \rfloor \leq a/b < \lfloor a/b \rfloor + 1$. Since $b > 0$, the left inequality together with the definition of c implies $c \geq 0$, and the right one implies $c < b$, that is, $c \leq |b| - 1$ and $c/b \geq 0$, since b is a positive integer.

Assume now that $b < 0$. Recall also that $\lfloor a/b \rfloor \leq a/b < \lfloor a/b \rfloor + 1$. Since $b < 0$, the left inequality implies $c \leq 0$, and the right one implies $c \geq b + 1$. Hence, since $b < 0$, we conclude that $|c| \in \{0, \ldots, |b| - 1\}$ and $c/b \geq 0$.

Since in both cases $c/b \geq 0$ and $|c| \in \{0, \ldots, |b| - 1\}$, we deduce that

$$c/b \in \{0, \ldots, (|b| - 1)/|b|\},$$

which ends the proof. □

Until now, we have seen how to rewrite the exact result $r = x^{1/n}$ as $r = \ell \cdot 2^d$, and we have given a definition for ℓ and the exponent d. Let us now determine the range in which the exact value ℓ lies.

Property 3.3. *For x a nonzero positive (sub)normal floating-point number defined in (1.5) in Section 1.2.1, the exact value ℓ satisfies:*

$$\ell \in [1,2) \quad \text{if } n \geq 0, \quad \text{and} \quad \ell \in (1,2] \quad \text{if } n < 0. \tag{3.14}$$

Proof. From (3.9) and (3.12), we know that $\ell = s \cdot (m'_x)^{1/n}$ if $n \geq 0$, and $\ell = s \cdot (2/m'_x)^{1/n}$ if $n < 0$. By (3.13), $s \in [1, 2^{(|n|-1)/|n|}]$. Besides, $m'_x \in [1, 2 - 2^{1-p}]$ implies $(m'_x)^{1/n} \in [1, 2^{1/n})$ if $n > 0$, and $(m'_x)^{1/n} \in (1, 2^{1/|n|}]$. The conclusion follows immediately. □

We conclude from (3.14) that $\circ(\ell) \in [1,2]$, for $\circ \in \{\mathrm{RN}_p, \mathrm{RU}_p, \mathrm{RD}_p, \mathrm{RZ}_p\}$, as required in Section 2.2.1.

Remark when $n = 2$. Moreover, we will see in Corollary 3.1 below that when $n = 2$, then the exact value ℓ in (3.12) is always strictly less than $2 - 2^{-p}$. Consequently, the correctly-rounded value of $\circ(\ell)$ may be equal to 2 only for $\circ = \mathrm{RU}_p$:

$$\text{when } n = 2, \quad \circ(\ell) \in \begin{cases} [1, 2 - 2^{1-p}] & \text{for } \circ \in \{\mathrm{RN}_p, \mathrm{RD}_p, \mathrm{RZ}_p\}, \\ [1,2] & \text{for } \circ = \mathrm{RU}_p. \end{cases}$$

Corollary 3.1. *When $n = 2$ the exact value of ℓ as in (3.12) satisfies*

$$1 \leq \ell < 2 - 2^{-p}.$$

Proof. By definition, $m'_x \in [1, 2 - 2^{1-p}]$ and $(m'_x)^{1/2} \in [1, 2 - 2^{1-p}]$. Assume first that $s = 1$: it is obvious to see that in this case, ℓ in (3.12) satisfies $\ell \in [1, 2 - 2^{1-p}]$.

Assume now that $s = 2^{1/2}$. We have $\ell \geq 2^{1/2} \geq 1$. Let us consider $\ell \geq 2 - 2^{-p}$: we know by definition that $(m'_x)^{1/2} \geq (2/2^{1/2}) \cdot (1 - 2^{-p-1})$. It follows that $m'_x \geq 2 \cdot (1 - 2^{-p} + 2^{-2p-2})$, and $m'_x \geq 2 - 2^{1-p} + 2^{-2p-1}$. However, we know that $m'_x \leq 2 - 2^{1-p}$. Hence, when $s = 2^{1/2}$, we have $1 \leq \ell < 2 - 2^{-p}$ as well, that ends the proof. □

This property on ℓ in the particular case $n = 2$ does not improve the method presented in Section 2.3.1 for packing the result. However, it is worth being noticed, since it can be used for implementations that return, for example the triple (sign, exponent, trailing significand) in three separated fields: in this case, an exponent and trailing significand update would indeed be needed if $\circ(\ell) = 2$.

3.1.3 Properties useful for correct rounding

This section gives now some properties that can be useful for implementing correct rounding, especially to determine if r can be exactly a floating-point number, or halfway between two floating-point numbers. Such properties are given in [IM99] for reciprocal, square root, and reciprocal square root. Here, we extend this approach to any function $x^{1/n}$, $n \in \mathbb{Z} \setminus \{0,1\}$. Table 3.1 below summerizes the properties presented in the remainder of the section.

	Exact point	Midpoint	Underflow
$n \geq 2$	yes (Property 3.4)	no (Property 3.6)	no (Property 3.1)
$n = -1$	no (Property 3.5)	no (Property 3.6)	yes (Property 3.2)
$n \leq -2$			no (Property 3.1)

Table 3.1: Summary of properties useful for rounding $x^{1/n}$ correctly (for non trivial input).

To do so, let us first define what we call a *trivial input*.

Definition 3.2 (trivial input). *Let x be a nonzero positive (sub)normal floating-point number. It is a* trivial input *if and only if the normalized significand m'_x equals* 1.

Can $x^{1/n}$ be exactly a floating-point number?

We know from the definition of the result $r = x^{1/n}$ in (3.3) that it is exactly a floating-point number if the value $\ell \cdot 2^{-\lambda_r}$ is representable with at most $p-1$ fraction bits, that is, if $\ell \cdot 2^{-\lambda_r}$ is a floating-point number in precision p. Let us now see if $\ell \cdot 2^{-\lambda_r}$ can be exactly a floating-point number. But before, let us define q as follows:

$$q \in \mathbb{N} \quad \text{such as} \quad \ell \cdot 2^{p-1-q} \quad \text{is an odd integer.} \tag{3.15}$$

More precisely, if the exact value ℓ is exactly a floating-point number, using the ranges for ℓ in (3.14), we obtain

$$q \in \{0, \ldots, p-1\} \quad \text{if } n > 0, \quad \text{and} \quad q \in \{0, \ldots, p\} \quad \text{if } n < 0.$$

• **First case:** $n \geq 2$. When $n \geq 2$, we will see in Property 3.4 below when the exact significand ℓ in (3.12) can be exactly a floating-point number (see Example 3.2). And since $\lambda_r = 0$ the exact result in (3.3) does not denormalize, we conclude that when $n \geq 2$, the exact result r may be exactly a floating-point number.

Property 3.4. *For x a nonzero positive (sub)normal floating-point number defined in (1.5) in Section 1.2.1 and $n \geq 2$, the exact value ℓ may be exactly a floating-point number if and only if*

$$0 \leq c + (p-1-q) \cdot n \leq p-1. \tag{3.16}$$

Proof. By definition, $m'_x \in [1, 2 - 2^{1-p}]$ and $\ell = 2^{c/n} \cdot (m'_x)^{1/n}$. Assume that ℓ is exactly a floating-point number: $\exists\, q \in \{0, \ldots, p-1\}$ such that $\ell \cdot 2^{p-1-q}$ is an odd integer. By definition of ℓ, we have

$$(\ell \cdot 2^{p-1-q})^n = \left((m'_x)^{1/n} \cdot 2^{c/n} \cdot 2^{p-1-q}\right)^n,$$

with $(\ell \cdot 2^{p-1-q})^n$ an odd integer. But $\left((m'_x)^{1/n} \cdot 2^{c/n} \cdot 2^{p-1-q}\right)^n = (m'_x) \cdot 2^{c+(p-1-q)\cdot n}$, thus ℓ is a floating-point number if and only if $(m'_x) \cdot 2^{c+(p-1-q)\cdot n}$ is an odd integer, that is, by definition of m'_x if and only if

$$0 \leq c + (p-1-q) \cdot n \leq p-1.$$

However, from (3.1), we have $0 \leq c \leq n-1$. Hence, we know that $0 \leq c + (p-1-q) \cdot n \leq n \cdot p - 1$. Thus, we can find some m'_x for which ℓ satisfies the condition (3.16). That is ℓ may be exactly a floating-point number, that ends the proof. □

Example 3.2. *Let us define m'_x as*

$$m'_x = 1.00000011000101011000000_2.$$

For $n = 3$ and $c = 2$, we have $\ell = 2^{2/3} \cdot (m'_x)^{1/3}$ and

$$\ell = 1.10011000000000000000000_2,$$

that is exactly a significand on $p-1$ fraction bits. Let us now verify that ℓ satisfies the condition (3.16). Here, we have $q = 18$, $c + (p-1-q) \cdot n = 17$, that satisfies (3.16).

This property makes the rounding procedure more complicated, especially for RoundToward-Positive or RoundTowardNegative, with negative or positive results, respectively, since the test $u > \ell$ is more complicated to be implemented in our context than $u \geq \ell$ (for example, see 3.5.6 for more details).

Let us now study the possible values that q can take so that the condition (3.16) remains satisfied. From Property 3.4, and more particularly from (3.16), we know that ℓ is exactly a floating-point number if and only if $0 \leq c + (p-1-q) \cdot n \leq p-1$, that is, if and only if the integer q satisfies:

$$\lceil (1 - p + n \cdot p - n)/n \rceil \leq q \leq p - 1. \tag{3.17}$$

For example, assume that $p = 24$. Table below shows the range of possible values q for various values n. We can observe that for $n \geq 24$, the single possible value q is 23. In this case, we have $\ell = 1.0$ and $m'_x = 1.0$.

Value of n	2	3	4	$\cdots$	23	24
Range for q	$12 \leq q \leq 23$	$16 \leq q \leq 23$	$18 \leq q \leq 23$	$\cdots$	$22 \leq q \leq 23$	$23 \leq q \leq 23$

More generally, we can observe that when $n \geq p$, we have:

$$p - 2 < (1 - p + n \cdot p - n)/n \leq p - 1 \quad (\text{when } n \geq p),$$

and in this case, the value q in (3.17) satisfies:

$$p - 1 \leq q \leq p - 1 \quad \text{that is} \quad q = p - 1.$$

From that statement, we can observe that the only real value ℓ that can be an exact floating-point number is $\ell = 1$, which is obtained with $m'_x = 1$ and $c = 0$. Hence, Remark 3.1 follows.

Remark 3.1. *Assume that x is a nonzero positive (sub)normal floating-point number defined in (1.5) in Section 1.2.1 and $n \geq 2$. If $n \geq p$, then the exact result $x^{1/n}$ is exactly a floating-point number if x is a* trivial input *as follows:*

$$x = 2^{e'_x}, \quad \textit{with} \quad e'_x \bmod n = 0.$$

This remark may be useful for implementing the function $x^{1/n}$ for $n \geq p$. In these cases, the inputs that lead to an exact floating-point number ℓ may be handled separately. Hence, the rounding $u > \ell$, may be replaced by $u \geq \ell$ in RoundTowardPositive and RoundTowardNegative, when the exact result is negative and positive, respectively, which is easier to be implemented on the ST231 processor.

• **Second case:** $n \leq -1$. In this case, Property 3.5 below says that, except for some trivial input, the exact value of ℓ cannot be exactly a floating-point number, and the exact result cannot be a floating-point number neither.

Property 3.5. *For x a nonzero positive (sub)normal floating-point number defined in (1.5) in Section 1.2.1 and $n \leq -1$, ℓ cannot be exactly a floating-point number, except for inputs of the form*

$$x = 2^{e'_x}, \quad \textit{with} \quad (e'_x + 1) \bmod n = n + 1.$$

Proof. By definition, $m'_x \in [1, 2 - 2^{1-p}]$ and $\ell = 2^{c-1/n} \cdot (m'_x)^{1/n}$. If ℓ is exactly a floating-point number and since $\ell \in [1, 2]$, then $\exists\, q \in \{0, \ldots, p\}$ such that $\ell \cdot 2^{p-1-q}$ is an odd integer. By definition of ℓ, we have

$$(\ell \cdot 2^{p-1-q})^n = \left((m'_x)^{1/n} \cdot 2^{(c-1)/n} \cdot 2^{p-1-q}\right)^n,$$

with $(\ell \cdot 2^{p-1-q})^n$ an odd integer. But $\left((m'_x)^{1/n} \cdot 2^{(c-1)/n} \cdot 2^{p-1-q}\right)^n = (m'_x) \cdot 2^{c-1+(p-1-q)\cdot n}$, thus ℓ is a floating-point number if and only if $(m'_x) \cdot 2^{c-1+(p-1-q)\cdot n}$ is an odd integer, that is by definition of m'_x if $0 \leq c-1+(p-1-q)\cdot n \leq p-1$. But by definition of $c \leq 0$ and $n \leq -1$, and since $q \leq p$, then we know that $c-1+(p-1-q)\cdot n \leq 0$. More particularly, we have $c-1+(p-1-q)\cdot n = 0$ if and only if $q = p$ and $c = n+1$. Otherwise $c-1+(p-1-q)\cdot n \leq -1$. Thus ℓ cannot be exactly a floating-point number, except when $q = p$ and $c = n+1$, that is when $\ell = 1.0$, and more generally when $m'_x = 1.0$ and $(e'_x + 1) \bmod n = n+1$: only for some trivial inputs. □

From Property 3.5, we can observe that for $n \leq -1$ the exact value is a floating-point number only for input of the form $x = \pm 2^{e'_x}$ with $(e'_x + 1) \bmod n = n+1$: in this case, $\ell = 1$. From (3.4) and (3.8), when $n \leq -1$, we know that $\lambda_r \in \{0, 1, 2\}$: in this case, if the exact value $\ell = 1$ then we conclude that the value $\ell \cdot 2^{-\lambda_r}$ is exactly representable with $p-1$ fraction bits.

In this case, this remark can also be used to simplify the rounding procedure for RoundTowardPositive and RoundTowardNegative when the exact result is negative and positive, respectively, while the case where the exact result is a floating-point number may be handled separately.

Can $x^{1/n}$ be halfway between two floating-point numbers?

It remains now to see if $x^{1/n}$ can be exactly halfway between two floating-point numbers, that is, if $\ell \cdot 2^{-\lambda_r}$ is representable with exactly p fraction bits. Let us consider the cases $|n| \geq 2$ and $n = -1$ separately.

• **First case:** $|n| \geq 2$. When $|n| \geq 2$, we know from the definition of the result $r = x^{1/n}$ in (3.3), since $\lambda_r = 0$, that r is exactly halfway between two floating-point numbers if and only if ℓ is representable with exactly p fraction bits, that is, ℓ is halfway between two floating-point numbers. We will show in Property 3.6 that the exact value ℓ, defined in (3.3) as $\ell = s \cdot (m'_x)^{1/n}$, cannot be exactly halfway two floating-point numbers.

Property 3.6. *For x a nonzero positive (sub)normal floating-point number defined in (1.5) in Section 1.2.1 and $|n| \geq 2$ or $n = -1$, the exact value ℓ cannot be exactly halfway between two floating-point numbers.*

Proof. By definition, $m'_x \in [1, 2 - 2^{1-p}]$ and $\ell = 2^{(c-\mu)/n} \cdot (m'_x)^{1/n}$. If ℓ is exactly halfway between two floating-point numbers, then $\ell_p = 1$ and $\ell = \ell_0.\ell_1\ell_2 \ldots \ell_{p-1}1$. By multiplying by 2^p and raising to the power n, we get

$$(\ell \cdot 2^p)^n = \left((m'_x)^{1/n} \cdot 2^{(c-\mu)/n} \cdot 2^p\right)^n. \tag{3.18}$$

Assume first $n \geq 0$. Here the left-hand side of (3.18) is an odd integer. However, in this case $\mu = 0$ and $c \geq 0$, and the right-hand side of (3.18) equals $(m'_x) \cdot 2^{c+p\cdot n}$. Using the fact that $p \geq 1$ and $n \geq 2$, we deduce that $c + p \cdot n > 2p$, and the right-hand side of (3.18) is an even integer. This yields a contradiction, and ℓ cannot be exactly halfway between two floating-point numbers.

Assume now $n < 0$, then $\mu = 1$ and $c \leq 0$. Thus, we can rewrite (3.18) as follows

$$(\ell \cdot 2^p)^{|n|} \cdot m'_x = 2^{|n|\cdot p - c + 1}. \tag{3.19}$$

It follows that the odd integer $\ell \cdot 2^p$ divides the right-hand side of (3.19). However, using $p \geq 1$ and $c \leq |n| - 1$, we conclude the righ-hand side of (3.19) is a positive power of two. Hence this yields also a contradiction, and ℓ cannot be exactly halfway between two floating-point numbers neither. □

Let us assume a value u that approximates ℓ such that $|\ell - u| < 2^{-p}$, as in (2.15) and since $\lambda_r = 0$. In the rounding algorithm presented in Section 2.2.4 for the RoundTiesToEven rounding-direction attribute, to handle the case where ℓ is exactly halfway between two floating-point numbers, we introduce the condition $u = \ell \ \wedge \ u_{p-1} = 0$. But since in the present case ($|n| \geq 2$), ℓ cannot be exactly halfway between two floating-point numbers, the rounding test of Section 2.2.4 for the RoundTiesToEven rounding-direction attribute can be simplified (and its implementation as well), as follows

$$\begin{aligned} &\text{if } u \geq \ell \text{ then} \\ &\quad m_r = \texttt{truncate}(u)_{p-1}, \\ &\text{else} \\ &\quad m_r = \texttt{truncate}(u + 2^{-p})_{p-1}. \end{aligned} \tag{3.20}$$

(Remark here that the rounding test $u \geq \ell$ is preferred to $u > \ell$, since here they are equivalent and the former is easier to be implemented.)

• **Second case:** $n = -1$. In this case, we know from Property 3.6 that the real value ℓ defined in (3.12) (for $n < 0$) can never be halfway between two floating-point numbers. In order to conclude that r cannot be halfway between two floating-point numbers neither, let us distinguish two cases:

- If $\lambda_r = 0$, then it is obvious that $\ell = \ell \cdot 2^{-\lambda_r}$, and $r = x^{-1}$ defined as $r = \ell \cdot 2^d$ cannot be exactly halfway between two floating-point numbers.
- If $\lambda_r \in \{1, 2\}$: in this case, r is exactly halfway between two floating-point numbers if ℓ is a floating-point number on $p - \lambda_r$ fraction bits. However from Property 3.5, ℓ cannot be exactly a floating-point number, more particularly a floating-point number on $p - \lambda_r$ fraction bits. It follows that $r = x^{-1}$ cannot be exactly a floating-point number neither.

Therefore, $n = -1$, the exact result $r = x^{-1}$ cannot be exactly halfway between two floating-point numbers. This property makes the rounding procedures simpler, especially for the RoundTiesToEven rounding-direction attribute, for which the rounding algorithm can be simplified as in (3.20).

3.2 Computation of a one-sided approximation

In this section, we discuss now how to approximate the real value ℓ defined in the previous section. As we have seen in Section 2.2.4, to ensure correct rounding, we may compute a *one-sided approximation* v of the exact value ℓ, defined as in [EL04]. The value v approximates ℓ from above, such that $-2^{-p} < \ell - v \leq 0$, which is implied by the more symmetric constraint

$$|(\ell + 2^{-p-1}) - v| < 2^{-p-1}. \tag{3.21}$$

By Property 3.3, $\ell \in [1, 2]$, thus $v \leq 3$ and its integer part can be represented using at most 2 bits:

$$v = (v_{-1} v_0 . v_1 v_2 \cdots v_{k-2})_2.$$

As we will see further, this form will in fact be the natural result of some derivations based on the triangular inequality. The main idea consists in considering the value $\ell + 2^{-p-1}$ to be approximated as the exact result of a particular function F, approximating this function by a suitable polynomial P, and then evaluating this polynomial P using an evaluation program $\mathcal{P}$. If the polynomial P is "accurate enough" with respect to the function F, and the evaluation of P entails a "small

enough" evaluation error, then we may ensure that the computed value v is "close enough" to the exact value $\ell+2^{-p-1}$, so that we are able to compute the correctly-rounded value $\circ(\ell)$ or $\circ(\ell\cdot 2^{-\lambda_r})$.

This approach is well-known (see [EL04, §8.6], for example) and, as we have seen in introduction, computing such an approximation v usually relies on *iterative*, *multiplicative*, or *polynomial-based* methods. In most cases, these methods are univariate methods, while the range reduction is handled through a last operation. However the novelty of our approach relies on the approximation of $\ell+2^{-p-1}$ by a suitable *bivariate polynomial*, where the range reduction is incorporated into the evaluation of this polynomial. The remainder of this section presents this approach in more detail.

3.2.1 Bivariate polynomial approximation

Definition of the bivariate polynomial

In this section we define the polynomial used to compute the value v that approximates the exact value ℓ from above as in (3.21). Recall that x is a nonzero positive (sub)normal floating-point number written: $x = m'_x \cdot 2^{e'_x}$ as usual (see for example Section 1.2.1). First, using the expression of ℓ in (3.12), we consider the value $\ell + 2^{-p-1}$ to be computed as the exact value $F(s^*, t^*)$, where

$$s^* = 2^{c/n} \quad \text{and} \quad t^* = m'_x - 1 \in \mathcal{T} = [0, 1 - 2^{1-p}],$$

and where

$$F : (s,t) \mapsto 2^{-p-1} + s \cdot f(t), \tag{3.22}$$

with $f(t)$ a univariate function defined, according to the sign of n, as follows

$$f : t \mapsto \begin{cases} (1+t)^{1/n}, & \text{if } n \geq 2, \\ 2^{-1/n} \cdot (1+t)^{1/n}, & \text{if } n \leq -1. \end{cases} \tag{3.23}$$

Example 3.3. *Let $n = 3$, $p = 24$, and x be a positive (sub)normal floating-point number. Then using (3.12),*

$$\ell = 2^{c/3} \cdot (m'_x)^{1/3}, \quad \textit{with} \quad c \in \{0, 1, 2\}.$$

Furthermore, the exact value $\ell + 2^{-25}$ can be seen as the exact result of the function

$$F : (s,t) \mapsto 2^{-25} + s \cdot (1+t)^{1/3},$$

at the point (s^*, t^*), with $s^* = 2^{c/3}$ and $t^* = m'_x - 1$.

Whatever the value of n, the function F cannot in general be evaluated directly on a computer. A second step thus consists in approximating F over $\mathcal{S} \times \mathcal{T}$ by a single bivariate polynomial P. We observe that when $n = -1$ then $c = 0$ and $s^* = 1$. Thus, the function F can be defined in a simpler way as $F : t \mapsto 2^{-p-1} + 2^{-1/n} \cdot (1+t)^{1/n}$, and in this case, we shall approximate F by a univariate polynomial. But for the generality of this approach, we consider that we approximate the function F by a bivariate polynomial $P(s,t)$, where the first variable enables to handle the range reduction, while the second depends on the input scaled significand m'_x. The interest of approximating the function F by a bivariate polynomial comes from the fact that a polynomial can be evaluated using only additions, subtractions, and multiplications, that are the only efficient instructions available on most processors, as the ST231 processor.

Since the function F is linear with respect to the variable s, we reduce the approximation to univariate approximation by taking

$$P(s,t) = 2^{-p-1} + s \cdot a(t), \tag{3.24}$$

with $a(t)$ a univariate polynomial of degree δ that approximates the function $f(t)$ in (3.23) over $\mathcal{T}$. This first step entails an *approximation error* denoted by $\alpha(a)$ and defined as the real number

$$\alpha(a) = \max_{t\in\mathcal{T}} |a(t) - f(t)|\,, \tag{3.25}$$

with $f(t)$ as in (3.23).

Definition 3.3 (*minimax*). *The* minimax *polynomial of degree δ with respect to the function $f(t)$ over $\mathcal{T}$ is the unique real polynomial $a^* \in \mathbb{R}[t]_\delta$, where $\mathbb{R}[t]_\delta$ denotes the set of univariate real polynomials of degree at most $\delta \in \mathbb{N}$, such that*

$$\alpha(a^*) \leq \alpha(a) \quad \textit{and} \quad a \in \mathbb{R}[t]_\delta.$$

(See [PT09] or [Ste98, p. 12] for details on the uniqueness of the minimax *polynomial.) This* best *polynomial approximant a^* may be approximated using Remez'algorithm [Rem34].*

The polynomial approximant is actually computed using Remez'algorithm of the software environment Sollya [Che09], [Lau08], [CL].

Making the polynomial coefficients and evaluation point machine-representable

The memory of a processor, and especially of the ST231, is finite and the coefficients of the polynomial approximant as well as the input point (s^*, t^*) have to be adjusted to be stored in finite precision k. Concerning the coefficients, the polynomial $a(t)$ is built in this sense, by computing an approximant polynomial whose all coefficients can be exactly representable using only k bits. (This polynomial approximant is computed using Remez' algorithm of Sollya.) Concerning the input (s^*, t^*), we see that t^* fits into a k-bit number, since

$$\begin{aligned} t^* &= m'_x - 1 \\ &= 0.m_{\lambda_x+1}\cdots m_{p-1}, \quad \text{with} \quad \lambda_x \geq 0 \quad \text{and} \quad p < k. \end{aligned}$$

However s^* may have an infinite binary expansion, or at least may require more than k bits to be represented, and it has to be rounded. Assuming $k = 32$, in Example 3.3, the value s^* belongs to $\{1, 2^{1/3}, 2^{2/3}\}$, that is,

$$\begin{aligned} s^* \in \{1,\ & \underbrace{1.01000010100010100011111001100}_{\text{32 bits}}0110\cdots_2\,, \\ & \underbrace{1.10010110010111111110101001010 01}_{\text{32 bits}}1110\cdots_2\}. \end{aligned}$$

By definition, we know that $s^* \in [1,2)$, and it cannot be stored in k-bit unsigned integer in precision k (with $k-1$ fraction bits). For $s \in \mathcal{S}$, let $\hat{s} = \mathrm{RN}_k$ be the rounded to nearest value of s:

$$|s - \hat{s}| \leq 2^{-k}. \tag{3.26}$$

We could improve the bound in (3.26) entailed by the rounding of s, by considering the fact that s can just take n different values. For example, for $(k,n) = (32,3)$, we can show that $|s-\hat{s}| < 2^{-32.25}$

(obtained for $s = 2^{1/3}$). However, this source of error is completely negligible compared to the other sources of error. Consequently, in pratice the bound in (3.26) is sufficient.

Now we should bound the error entailed on the evaluation by the rounding of input s. To do so let $\gamma(s) = \max_{(s,t)\in\mathcal{S}\times\mathcal{T}} |P(s,t) - P(\hat{s},t)|$ be the error on the evaluation due to the rounding of input s. By definition of the polynomial P, we deduce that $|P(s,t) - P(\hat{s},t)| = |s - \hat{s}| \cdot |a(t)|$ and

$$\gamma(s) = \max_{(s,t)\in\mathcal{S}\times\mathcal{T}} \left(|s - \hat{s}| \cdot |a(t)|\right). \tag{3.27}$$

Polynomial evaluation

Once we have built the bivariate polynomial P and made the input machine representable, it remains to write a program $\mathcal{P}$ to evaluate P at the point $(\hat{s}^*, t^*)$. Since the evaluation is done by a finite precision evaluation program, it entails a third source of error, called *evaluation error* and defined as

$$\rho(\mathcal{P}) = \max_{(\hat{s},t)\in\hat{\mathcal{S}}\times\mathcal{T}} |P(\hat{s},t) - \mathcal{P}(\hat{s},t)|\,. \tag{3.28}$$

Property 3.7. *Given ℓ, v, a and $\alpha(a)$, $\mathcal{P}$ and $\alpha(\mathcal{P})$, if the condition*

$$2^{1-1/|n|}\alpha(a) + \gamma(s) + \rho(\mathcal{P}) < 2^{-p-1} \tag{3.29}$$

is satisfied then (3.21) holds.

Proof. At each point $(s^*, t^*) \in \mathcal{S} \times \mathcal{T}$, we may check using the triangular inequality that

$$\begin{aligned} \left|\ell + 2^{-p-1} - v\right| &= |F(s^*,t^*) - \mathcal{P}(s^*,t^*)| \\ &\leq |F(s^*,t^*) - P(s^*,t^*)| + |P(s^*,t^*) - P(\hat{s}^*,t^*)| + |P(\hat{s}^*,t^*) - \mathcal{P}(\hat{s}^*,t^*)| \\ &\leq 2^{1-1/|n|}\alpha(a) + \gamma(s) + \rho(\mathcal{P}). \end{aligned}$$

But, from (3.21), the overall error has to be strictly less than 2^{-p-1}. Thus, if we can ensure that

$$2^{1-1/|n|}\alpha(a) + \gamma(s) + \rho(\mathcal{P}) < 2^{-p-1},$$

then (3.21) holds. □

In short, this approach consists in two main steps: finding a polynomial approximant $a(t)$ approximating the function $f(t)$ over $\mathcal{T}$, and an evaluation program $\mathcal{P}$ evaluating $P = 2^{-p-1} + s \cdot a(t)$, such that the condition (3.29) defined above is satisfied.

3.2.2 Certified approximation and evaluation error bounds

Now in this section, we will give some sufficient certified bounds on the approximation and evaluation errors in (3.25) and (3.28), so that if we manage to compute a polynomial P and an evaluation program $\mathcal{P}$ satisfying these conditions and such as the condition (3.29) holds, we are sure that the computed value v is close enough to the value ℓ in the sense of (3.21).

Approximation error bound

We know by definition that the rounding input and evaluation errors are non negative: $\gamma(s) \geq 0$ and $\rho(\mathcal{P}) \geq 0$. Thus, from the required condition (3.29), we deduce that the approximation error $\alpha(a)$ of the polynomial approximant $a(t)$ with respect to the function $f(t)$ over $\mathcal{T}$ has to satisfy $2^{1-1/|n|}\alpha(a) < 2^{-p-1} - \gamma(s)$, that is,

$$\alpha(a) < (2^{-p-1} - \gamma(s))/2^{1-1/|n|}. \tag{3.30}$$

In practice, we will compute a polynomial $a(t)$ together with a certified approximation bound θ, such as

$$\alpha(a) \leq \theta \quad \text{and} \quad \theta < (2^{-p-1} - \gamma(s))/2^{1-1/|n|}, \tag{3.31}$$

with θ a dyadic number and $\gamma(s)$ defined in (3.33) in the next paragraph.

Property 3.8 below bounding the range of $a(t)$ when (3.31) holds will be useful for bounding $\gamma(s)$ in the next paragraph.

Property 3.8. *Assume $n \geq |2|$. Given the polynomial approximant a defined above, for $t^* \in [0, 1 - 2^{1-p}]$, we have*

$$0 < a(t) < \sqrt{2} \quad \text{if } n \geq 2, \quad \text{and} \quad 0 < a(t) < 2 \quad \text{if } n \leq -2. \tag{3.32}$$

Proof. Assume first that $n \geq 2$. Consider the case when $n = 2$. Using (3.25) together with (3.31), we know that $(1+t)^{1/2} - 2^{-p-3/2} < a(t) < (1+t)^{1/2} + 2^{-p-3/2}$. Thus, using $t \geq 0$ gives $a(t) > 1 - 2^{-p-3/2}$. Then using $t \leq 1 - 2^{1-p}$ gives $a(t) < (2 - 2^{1-p})^{1/2} + 2^{-p-3/2} \leq 2^{1/2} - 2^{-p-3/2}$, since $2 - 2^{1-p} \leq 2 - 2^{1-p} + 2^{-2p-1}$ with $2 - 2^{1-p} + 2^{-2p-1} = (2^{1/2} - 2^{-p-1/2})^2$ together with the fact that the function $x \mapsto x^{1/2}$ is monotonically increasing. Consider now that $n > 2$. Here we have $a(t) > 1 - 2^{-p-3/2}$ (since $t \geq 0$). Using $t \leq 1 - 2^{1-p}$, as $(2 - 2^{1-p})^{1/n} < (2 - 2^{1-p})^{1/2}$, we conclude that $a(t) \leq 2^{1/2} - 2^{-p-3/2}$.

Assume now that $n \leq -2$. Using (3.25) together with (3.31), we know in this case that $2^{1/2} \cdot (1+t)^{-1/2} - 2^{-p-3/2} < a(t) < 2^{1/2} \cdot (1+t)^{-1/2} + 2^{-p-3/2}$. Thus, using $t \geq 0$ gives $a(t) < 2^{1/2} + 2^{-p-3/2}$, and then, using $t \leq 1 - 2^{1-p}$ gives $a(t) > 1 - 2^{-p-3/2}$, that ends the proof. □

Actually, the bounds in (3.32) on $a(t)$ are pessimistic, independent of n and the precision p. But in practice, they are sufficient for expressing the bound on $\gamma(s)$, since they will be multiplied by 2^{-k} which is tiny.

Polynomial input error bound

Recall that for $n = -1$, the value of s is always 1, so that $s = \hat{s}$ and $\gamma(s) = 0$ in (3.27). Otherwise, that is, when $|n| \geq 2$, we deduce from (3.26) and (3.32) in Property 3.8 an upper bound on the loss of accuracy when rounding input, defined as follows

$$\gamma(s) < \begin{cases} 2^{1/2-k} & \text{if } n \geq 2, \\ 2^{1-k} & \text{if } n \leq -2. \end{cases} \tag{3.33}$$

Evaluation error bound

Assume now that we have a polynomial $a(t)$ such as (3.31). From (3.29) above, the evaluation error has to be such that the overall error is strictly less than 2^{-p-1}, that is, the evaluation program $\mathcal{P}$ has to be such that

$$\gamma(s) + \rho(\mathcal{P}) < 2^{-p-1} - 2^{1-1/|n|} \cdot \theta. \tag{3.34}$$

Recall that if $n = -1$, then $s = \hat{s}$ and $\gamma(s) = 0$. Consequently, the evaluation error $\rho(\mathcal{P})$ simply has to be such that

$$\rho(\mathcal{P}) < 2^{-p-1} - \theta \qquad \text{if } n = -1. \tag{3.35}$$

When $|n| \geq 2$, it follows from (3.33) and (3.34) that

$$\rho(\mathcal{P}) \leq \begin{cases} 2^{-p-1} - 2^{1-1/|n|} \cdot \theta - 2^{1/2-k}, & \text{if } n \geq 2, \\ 2^{-p-1} - 2^{1-1/|n|} \cdot \theta - 2^{1-k}, & \text{if } n \leq -2. \end{cases} \tag{3.36}$$

These results are summarized in Table 3.2 below.

Condition on n	Evaluation error bound
$n \geq 2$	$\rho(\mathcal{P}) \leq 2^{-p-1} - 2^{1-1/\|n\|} \cdot \theta - 2^{1/2-k}$
$n = -1$	$\rho(\mathcal{P}) < 2^{-p-1} - \theta$
$n \leq -2$	$\rho(\mathcal{P}) \leq 2^{-p-1} - 2^{1-1/\|n\|} \cdot \theta - 2^{1-k}$

Table 3.2: Evaluation error bound, according to n.

We aim at giving certified error bounds. And since $2^{1-1/|n|}$ may have an infinite binary expansion, the bound $2^{-p-1} - 2^{1-1/|n|} \cdot \theta - \gamma(s)$ may not be evaluated exactly, and may not be certified. Thus let us define `RoundDownward`$(\cdot)$ a function that computes a rounding downward of a given value. Let finally η be a certified evaluation error bound, defined as follows

$$\eta = \begin{cases} \texttt{RoundDownward}(2^{-p-1} - 2^{1-1/|n|} \cdot \theta - 2^{1/2-k}) & \text{if } n \geq 2, \\ \texttt{RoundDownward}(2^{-p-1} - \theta) & \text{if } n = -1. \\ \texttt{RoundDownward}(2^{-p-1} - 2^{1-1/|n|} \cdot \theta - 2^{1-k}) & \text{if } n \leq -2, \end{cases} \tag{3.37}$$

with η a dyadic number.

Once we have written an evaluation program $\mathcal{P}$, we just have to check that $\rho(\mathcal{P}) \leq \eta$ (for $n \neq -1$) or $\rho(\mathcal{P}) < \eta$ (for $n = -1$), to ensure that the overall error is strictly less than 2^{-p-1} and thus ensure correct rounding. Here, we observe that the accuracy sufficient for the evaluation depends on the accuracy of the polynomial approximant $a(t)$. And, the more accurate the polynomial $a(t)$ is with respect to the function $f(t)$ over $\mathcal{T}$, the less accurate the evaluation program $\mathcal{P}$ might be.

3.2.3 Automatic generation of polynomial coefficients

Description of the automatic process

This section presents an automatic approach for generating the coefficients of the polynomial $a(t)$ together with the sufficient error bounds θ and η. Indeed, we deduced that the implementation of the nth roots, and their reciprocals, relies (for given values n) on the approximation of a particular function F defined in (3.22) by a bivariate polynomial P defined in (3.24). Once we have computed the coefficients of the polynomial approximant $a(t)$ and the sufficient error bounds θ and η, the main part will be to write an accurate enough evaluation program, that is, satisfying the bound η in (3.37).

Hence, we can already deduce a methodology for generating the coefficients of $a(t)$ and the certified bounds θ and η in an automatic way. This automatic generation process, implemented as a Sollya script, is presented in Listing 3.1 below. From the value n, we compute an estimation of the minimal degree δ of $a(t)$ using the function `guessdegree` of Sollya in line 24. Then we

compute $a(t)$ by truncating on the same format each coefficient of the Remez'polynomial of degree δ, in line 35. Here, all the coefficients are represented in the same format in absolute value (without bit handling the sign), while their signs will be handled through an appropriate choice of arithmetic operator (see Section 3.5.4 below, or paragraph "Unsigned fixed-point evaluation and arithmetic operator choice" of Section 5.1.1 for examples). Once we know the polynomial approximant $a(t)$, it simply remains to compute the certified error bound θ in lines 38 and 39 and check if it satisfies (3.31) (if it is not the case, we increase δ). Finally, the bound η is computed using a certified supremum norm of Sollya (see [CL07] or [CJL09]) and (3.37), in line 50.

Example 3.4 displays the result of an execution of the script in Listing 3.1, for $(k, p, n) = (32, 24, 3)$, that is, for implementing the cube root in *binary32* floating-point arithmetic. The output of this script contains in particular the coefficients of the polynomial approximant $a(t)$, the certified error bounds, and the possible values of $\hat{s}^*$. From now, the full implementation can be obtained, in an automatic way, using the basic blocks presented in Chapter 2. It simply remains:

1. to compute the exponent D and the value c, as shown in Section 3.3.2;
2. to select the input s^*, among those returned by the script;
3. to handle special inputs, as presented in Section 3.4;
4. to generate automatically an efficient polynomial evaluation program, as explained in Part II;
5. to implement the rounding condition.

The selection of the input s^* may be done naively by testing the value c and deducing the corresponding input s^*. Therefore the last part to be automated is the implementation of rounding condition, so that the correct rounding can be computed. Currently, this part is based on the inversion of the function, and is done separately for each function. Two examples are given in Sections 3.5.6 and 3.6.5, for square root and reciprocal, respectively.

Example 3.4. *Assume $n = 3$, $(k, p) = (32, 24)$, as defined in Example 3.3. Using the script in Listing 3.1 by calling* `nroot(32,24,3);`*, the results are the following.*

```
----------------------------------------------------
Register size k:     32
Working precision p: 24

Value n:                    3
Function f:                 (1 + x)^(1 / 3)
Interval T:                 [0;0.99999988079071044921875]
Approximation error bound: (2.98023223876953125e-8 - 2^(-3.15e1))/2^(1-1/3)
Polynomial input error bound: 2^(-3.15e1)

Required degree (delta): 8
Coefficient format:      Q31
Coefficients:
 a0 = 0x80000008; // (+) 0x80000008p-31 = 1.00000000372529029846191406250
 a1 = 0x2aaaa4ce; // (+) 0x2aaaa4cep-31 = 0.33333263453096151351928710937500
 a2 = 0x0e383a31; // (-) 0x0e383a31p-31 = 0.11109092133119702339172363281250
 a3 = 0x07df4484; // (+) 0x07df4484p-31 = 6.15010876208543777465820312500e-2
 a4 = 0x05197b56; // (-) 0x05197b56p-31 = 3.98401422426104545593261718750e-2
 a5 = 0x034b760a; // (+) 0x034b760ap-31 = 2.57403897121548652648925781250e-2
 a6 = 0x01c8cb96; // (-) 0x01c8cb96p-31 = 1.39402849599719047546386718750e-2
 a7 = 0x00a79101; // (+) 0x00a79101p-31 = 5.11372135952115058898925781250e-3
 a8 = 0x001d5785; // (-) 0x001d5785p-31 = 8.95442906767129898071289062500e-4
```

```
Value for s:
 c =  0 -> s = 0x80000000
 c = +1 -> s = 0xa14517cc
 c = +2 -> s = 0xcb2ff52a

Sufficient bounds
     theta: 145768613651649895977382 7101 * 2^(-118)
            ~ 2^(-2.776426132059602323273293212127 9951964e1)
     eta:   100397119873658572396549119622 1122347 * 2^(-145)
            ~ 2^(-2.540487070125620435548349606410 5085556e1)
------------------------------------------------
```

We deduce from the execution that for the implementation of cube root in the binary32 *format, the polynomial $a(t)$ is a degree-8 polynomial that approximates the function $f(t) = (1+t)^{1/3}$ with an approximation error $\alpha(a) \leq \theta \approx 2^{-27.76}$, which is actually strictly less than $(2^{-25} - 2^{-31.5})/2^{2/3} \approx 2^{-25.68}$. Also the evaluation of the polynomial P has to entail an evaluation error $\rho(\mathcal{P}) \leq \eta \approx 2^{-25.40}$.*

Remark 3.2. *Remark here that the* `RoundDownward` *function is implemented using Sollya, and consists in evaluating the expression by interval arithmetic in the current precision, and returning the lower bound of the resulting interval.*

Some numerical examples for the *binary32* format

Using the script presented in Listing 3.1 below, we can generate, in a faster way, the polynomial coefficients as well as the certified error bounds θ and η, for several values of n. Table 3.3 below shows, for $-4 \leq n \leq 4$ and $n \notin \{0, 1\}$, the degree δ of the polynomial approximant $a(t)$ and the bounds θ and η. These results have been generated in about 20s.

We can observe that for these small values of n, the polynomial is of degree between 8 and 10. For larger values of n, the polynomial approximant tends to be of smaller degree (of degree $\delta = 7$ for $n = 50$, of degree $\delta = 6$ for $n = 100$). Therefore, for larger values n, the selection of the input s^* and the implementation of the rounding condition will dominate the cost of the full implementation, since the polynomial evaluation will not be of larger degree.

Value of n	Degree δ	Approximation error bound θ	Evaluation error bound η
-4	8	$\approx 2^{-25.95}$	$\approx 2^{-28.09}$
-3	9	$\approx 2^{-27.76}$	$\approx 2^{-25.41}$
-2	9	$\approx 2^{-26.60}$	$\approx 2^{-25.94}$
-1	10	$\approx 2^{-26.39}$	$\approx 2^{-25.69}$
2	8	$\approx 2^{-27.99}$	$\approx 2^{-25.30}$
3	8	$\approx 2^{-27.76}$	$\approx 2^{-25.40}$
4	8	$\approx 2^{-27.74}$	$\approx 2^{-25.43}$

Table 3.3: Degree of the polynomial approximant $a(t)$, and approximation of the certified error bounds θ and η, for various values of n, and for $(k, p) = (32, 24)$.

3.2.4 Evaluation of the bivariate polynomial

Once we have computed the polynomial approximant using the script presented in Listing 3.1, we have seen that it remains to write an evaluation program $\mathcal{P}$ that evaluates the polynomial P, with

```
nroot = proc(k,p,n){
  // Definition of the function f, according to the sign of n
  if( n < 0 ) then {
    f = 2^(-1/n)*(1+x)^(1/n);
  } else {
    f = (1+x)^(1/n);
  };

  // Definition of the interval T
  T = [0,1-2^(1-p)];

  // Determination of the polynomial input error bound
  gamma = 0;                    // n  = -1
  if( n >= 2 ) then {
    gamma = 2^(1/2-k);          // n >=  2
  } else if ( n <= -2 ) then {
    gamma = 2^(1-k);            // n <= -2
  };

  // Computation of the approximation error bound, defined in (3.30)
  approx = (2^(-p-1)-gamma)/2^(1-1/abs(n));

  // Determination of the minimal degree δ=delta
  dinterval = guessdegree(f,T,approx);    // Sollya's guessdegree function
  delta = inf(dinterval);

  minimal = 0; while( minimal == 0 ) do {
    // Computation of the Remez's polynomial approximant
    astar = remez(f,delta,T,1,1e-7);        // Sollya's remez function

    // Determination of the size of the integer part of the coefficient
    Qf = k - GetIntegerPartSize(astar,n,p);

    // Truncation of each coefficient on Qf fraction bits
    a = TruncatePoly(astar,Qf);    // returns the truncated Reme'z polynomial

    // Computation of the certified error bound θ=theta in (3.31)
    diam=1e-8!; thetainterval = infnorm(f-a,T);  // Sollya's infnorm function
    theta = sup(thetainterval);

    // Checking if theta satisfies the condition in condition in (3.31)
    if( theta >= approx ) then {
      delta = delta + 1;
    } else {
      minimal = 1;
    };
  };

  // Computation of the certified evaluation error bound η=eta in (3.37)
  eta = RoundDownward(2^(-p-1) - 2^(1-1/abs(n))*theta - gamma);

  // Computation of each possible value of s
  lists = [||];
  for c from 0 to abs(n)-1 do
  {
    s = RNnroot(c,n,k-1); // returns RN_k(2^{c/|n|}), rounding to nearest
    lists = lists:.s;     // of 2^{c/|n|} on k-1 fraction bits
  };

  return [|k,p,n,f,T,approx,gamma,delta,astar,a,Qf,lists,theta,eta|];
};
```

Listing 3.1: Sollya script for automatic generation of polynomial coefficients.

an evaluation error satisfying the evaluation error bound η. Recall that we have to evaluate this polynomial at runtime. Of course, we have chosen a polynomial approximant of smallest degree, but we still have to try to evaluate it as efficiently as possible.

Among the most classical ways to evaluate the polynomial P, let us first quote Horner's rule [Knu98, §4.6.4], [Rev06, §1.3.1]. It evaluates the polynomial in a fully sequential way. Other classical methods enable to reduce the evaluation latency by exposing more instruction-level parallelism: the second-order Horner's rule [Knu98, §4.6.4] or the Estrin's method [Knu98, §4.6.6], [Rev06, §1.3.2]. This is usually done to the detriment of an increase of the total number of operations, especially of multiplications. These schemes are well-adapted for evaluating univariate polynomials, and the evaluation of the polynomial P usually relies on the evaluation of $a(t)$ and a last Horner's step (multiplication by s and addition). A first improvement may consists in distributing the multiplication by s over the evaluation of $a(t)$. The interest of using these schemes in fixed-point arithmetic will be presented in Chapter 5 through detailed implementation examples.

Others methods enable to evaluate a given polynomial by reducing the number of operations, and more particularly the number of multiplications. This is done by adapting the coefficients of the polynomial to be evaluated. Among these, let us quote, for example, Knuth and Eve's algorithm [Knu98, §4.6.4, Theorem E], [Eve64] and Paterson and Stockmeyer's algorithm [PS73]. However, we will see in Chapter 5 that these algorithms are not well-adapted for evaluating polynomials on fixed-point arithmetic, and that the adaptation of coefficients may lead to a loss of accuracy.

We will see in Chapter 5 that we can evaluate the polynomial P in a more efficient way than the classical methods, that is, with evaluation programs with smaller evaluation latency, especially on ST231 processor. The problem is that there is a very large number of evaluation programs for evaluating a given polynomial, even for small degrees (see Chapter 5). Therefore, in the remainder of this section, we will consider that we evaluate the polynomial P using a *best* evaluation program, that is, that reduces the evaluation latency and that is accurate enough. This *best* evaluation program can be obtained by using the tool and methodology presented in Chapter 6, integrated into CGPE.

3.3 Sign and exponent result implementation

In this section, we describe how to implement the computation of the sign and the exponent of the result from the input x, and more particularly from the k-bit unsigned integer X encoding x. Recall from (2.21) that the returned k-bit unsigned integer R that encodes the correctly-rounded result $\circ(r)$ is defined as

$$R = S_r + D \cdot 2^{p-1} + M_r. \tag{3.38}$$

3.3.1 Sign result computation

The computation of the sign s_r of the result is trivial since $s_r = s_x$ if n is odd, and $s_r = 0$ if n is even. Let S_r be the unsigned integer encoding s_r such that $S_r = s_r \cdot 2^{k-1}$. Then

$$S_r = \begin{cases} 0, & \text{if } n \text{ is even,} \\ X \wedge 2^{k-1}, & \text{if } n \text{ is odd.} \end{cases}$$

Assuming that n is odd, for $(k, p, e_{\max}) = (32, 24, 127)$, the computation of the sign of the result may be implemented with the following piece of C code.

```
Sr = X & 0x80000000;
```

3.3.2 Exponent result computation

Now let us have a look at how to implement the computation of the integer D in (3.38). As in Section 2.3 which discusses how to pack the correctly-rounded result, let E_r be the k-bit unsigned integer encoding the biased value of the result exponent e_r. Given n_r the "is normal bit" of the exact result r ($n_r = 1$ if the result is normal, and 0 otherwise), let D be the k-bit unsigned integer such that $D = E_r - n_r$. By definition of the interchange encoding presented in Section 1.2.2, we know that $E_r = e_r - e_{\min} + n_r$, and it follows that

$$D = e_r - e_{\min}.$$

When $|n| \geq 2$

In this first case, we know from Property 3.1 that the exact result never denormalizes and the exponent e_r is equal to the exponent $d = \lfloor (e'_x + \mu)/n \rfloor$ defined in (3.11). Thus, we have $D = \lfloor (e'_x - n \cdot e_{\min} + \mu)/n \rfloor$. Now recall that

$$e'_x = e_x - \lambda_x, \quad \text{with} \quad e_x = E_x + e_{\min} - n_x \quad \text{and} \quad \lambda_x = MX - w. \tag{3.39}$$

We then deduce

$$D = \lfloor (-1)^\mu \cdot (E_x - n_x - MX + C)/|n| \rfloor, \quad \text{with} \quad C = (1-n) \cdot e_{\min} + w + \mu. \tag{3.40}$$

Notice that once the value of n and the floating-point parameters are fixed, C is a constant (and thus independent of x).

When $n = -1$

In this case, we know that $e_r = \max(e_{\min}, d)$, and, from (3.11), $d = -e'_x - 1$. Consequently, $D = \max(0, -e'_x - e_{\min} - 1)$. Using as before the identity (3.39), we obtain

$$D = \max(0, -(E_x - n_x - MX + C)), \tag{3.41}$$

with C as in (3.40) for $n = -1$ (and $\mu = 1$), that is, $C = 2e_{\min} + w + 1$.

Implementation of integer division and its remainder

Once E_x, n_x, MX, and the constant C are known, the most difficult part is to implement the integer division by $|n|$. Here, we will thus consider only the case $|n| \geq 2$. (For $n = -1$, the computation of the exponent D is detailed in Section 3.6.2 below.) More particularly, we consider here the case when $|n|$ is not a power of two. If $|n|$ is a power of two, the implementation of integer division can be simply implemented with a shift in integer arithmetic (at least if $|n| < 2^k$), as shown in Section 3.5.2 for square root.

Moreover, by definition of $D = e_r - e_{\min}$ together with $e_r \in \{e_{\min}, \ldots, e_{\max}\}$, it follows that $D \geq 0$, and thus $(-1)^\mu \cdot (E_x - n_x - MX + C) \geq 0$ in (3.40) also. For clarity, we denote this integer by D_n, as follows:

$$D_n = (-1)^\mu \cdot (E_x - n_x - MX + C) \quad \text{with} \quad 0 \leq D_n < 2^k/|n|. \tag{3.42}$$

Let us now see how to implement the computation of $\lfloor D_n/|n| \rfloor$. Using the fact that $D_n < 2^k/|n|$ enables to gain some instructions, compared to what is proposed in [Jr.03] or implemented in the

ST200 compiler, since it can be done by simply using the multiplication function `mul`. Actually, let N be the k-bit unsigned integer encoding of $1/|n|$, such that:

$$N = \lceil 2^k/|n| \rceil \quad \text{with} \quad 0 \leq \lceil 2^k/|n| \rceil \cdot 2^{-k} - 1/|n| < 2^{-k}.$$

It follows that

$$D_n/|n| \leq (D_n \cdot N)/2^k < D_n/|n| + 2^{-k} \cdot D_n,$$

Then we can show (see [JR09b, Theorem 4.1]) that, if $n^2 \leq 2^p$,

$$\begin{aligned} \lfloor D_n/|n| \rfloor &= \lfloor (D_n \cdot N)/2^k \rfloor. \\ &= \texttt{mul}(D_n, N). \end{aligned} \tag{3.43}$$

Assuming as in Example 3.4 that $n = 3$ and $p = 24$, we deduce that $N = 0\text{x}55555556_{16}$. Hence, for $(k, p, e_{\max}) = (32, 24, 127)$, the implementation of D can be done as follows:

```
1  Dn = Ex - nx - MX + C;
2
3  D = mul(Dn , 0x55555556);
4
5  c  = Dn - D * 3;
```

Listing 3.2: Exponent D computation for $n = 3$ and $(k, p, e_{\max}) = (32, 24, 127)$.

It remains now to see how to compute the value c defined in (3.10) as $c = (e'_x + \mu) \bmod n$. Recall that $D = \lfloor (e'_x + \mu)/n \rfloor - e_{\min}$. Using this together with Definition 3.1, we deduce that

$$\begin{aligned} c &= (e'_x + \mu) - n \cdot \lfloor (e'_x + \mu)/n \rfloor \\ &= (e'_x + \mu) - n \cdot (D + e_{\min}). \end{aligned} \tag{3.44}$$

Using $E_x = e_x - e_{\min} + n_x$, and $e'_x = e_x - \lambda_x$ with $\lambda_x = MX - w$, it follows

$$c = (E_x - n_x - MX + C) - n \cdot D. \tag{3.45}$$

From Lemma 3.1, we know that $c/n \geq 0$. Hence, to compute $s^* = 2^{c/n}$, we will prefer to extract $|c|$ and compute $s^* = 2^{|c|/|n|}$. The interest of this approach is that it enables to write the same extraction code for $|n|$ and $-|n|$.

And finally, from (3.42), we have

$$|c| = D_n - |n| \cdot D. \tag{3.46}$$

For $n = 3$, (3.45) is equivalent to (3.46), and the remainder c of the integer division in computed in line 5 (Listing 3.2), and will be used for the selection of the input s^*. Remark that the extraction of the value c may be done in a more efficient way when n is a power of two, as presented in 3.5.3.

3.4 Implementation of special input handling

Until now we have seen how to compute nth roots and their reciprocals, for given values n and for general input x. Before describing two implementation examples in Section 3.5 and 3.6, we present in this section an efficient way of handling special input x.

Let x be a floating-point datum. From Properties 3.1 and 3.2, we know that the exact result of $x^{1/n}$ never under/overflows, and similarly for the correctly-rounded value, except for $n = -1$. Here, we consider the cases where $n \neq -1$, while this special case is presented in Section 3.6.3.

The remainder of this section is divided into three parts. First, we give the definition of what we call *special input* for the function $x^{1/n}$ according to the sign and the parity of n; then we present an efficient way to decide whether a floating-point datum x is a special input or not; and finally, we show how to compute the output prescribed by the IEEE 754-2008 standard.

3.4.1 Definition of special operands for $x^{1/n}$

Consider first the function $x^{1/n}$ for $n \geq 2$. Table 3.4 below shows the behavior of this function defined as the function `rootn` of the IEEE 754-2008 standard [IEE08, p. 44] for a given $n > 0$ and a given rounding-direction attribute $\circ$.

Input x	$+0$	-0	$+\infty$	$-\infty$	$x > 0$	$x < 0$	NaN
Result r when $n = 2$	$+0$	-0	$+\infty$	qNaN	$\circ(x^{1/n})$		qNaN
Result r when n is even and $\neq 2$		$+0$		$+\infty$	qNaN	$\circ(x^{1/n})$	qNaN
Result r when n is odd	$+0$	-0	$+\infty$	$-\infty$		$\circ(x^{1/n})$	qNaN

Table 3.4: Special values for $x^{1/n}$ for $n \geq 2$.

Let us now consider that $n \leq -1$. In this case, Table 3.5 below shows the behavior of this function, defined as the function `rootn` in the standard [IEE08, p. 44] for a given $n < 0$ and a given rounding-direction attribute $\circ$.

Input x	$+0$	-0	$+\infty$	$-\infty$	$x > 0$	$x < 0$	NaN
Result r when $n = -2$	$+\infty$	$-\infty$	$+0$	qNaN	$\circ(x^{1/n})$		qNaN
Result r when n is even and $\neq -2$		$+\infty$		$+0$	qNaN	$\circ(x^{1/n})$	qNaN
Result r when n is odd	$+\infty$	$-\infty$	$+0$	-0		$\circ(x^{1/n})$	qNaN

Table 3.5: Special values for $x^{1/n}$ for $n \leq -1$.

Therefore, the input x will be said "special" when

- n is even and $x \in \{x < 0, \pm 0, \pm\infty, \text{NaN}\}$,
- or n is odd and $x \in \{\pm 0, \pm\infty, \text{NaN}\}$.

In all these cases, the IEEE 754-2008 standard [IEE08] specifies that a special value to be returned. It remains now to see how to decide if a given input x is "special", and in this case what result shall be returned.

Remark here that the functions $x^{1/2}$ and $x^{-1/2}$ are defined as the square root (squareRoot) and the reciprocal square root (rSqrt), respectively, in the IEEE 754-2008 standard.

3.4.2 How to filter out special input?

Let X be the k-bit unsigned integer encoding the floating-point datum x, assuming the binary interchange encoding presented in Section 1.2.2.

Detecting whether x belongs to $\{x < 0, \pm 0, \pm\infty, \text{NaN}\}$

To detect if $x \in \{x < 0, \pm 0, \pm\infty, \text{NaN}\}$, let us consider the unsigned integer X encoding the floating-point datum x. From Table 1.4 and for $(k, p, e_{\max}) = (32, 24, 127)$, we can deduce the following piece of C code.

```
if(X > 0x80000000){ /* x < 0 */ }
else if((X & 0x7FFFFFFF) == 0x00000000){ /* +0 / -0 */ }
else if((X & 0x7FFFFFFF) == 0x7F800000){ /* +inf / -inf */ }
else if((X & 0x7FFFFFFF) >  0x7F800000){ /* NaN */ }
```

Notice however that this method does not use the binary interchange encoding as much as possible. To do so, let us observe that x is special input if and only if $X = 0$ or $X \geq 2^{k-1} - 2^{p-1}$. And since addition and subtraction are done modulo 2^k, the above condition turns out to be equivalent to the following one:

$$X - 1 \geq 2^{k-1} - 2^{p-1} - 1.$$

For $(k, p, e_{\max}) = (32, 24, 127)$, this second condition yields a code that is much simpler:

```
if((X - 1) >= 0x7F7FFFFF){
  /* x < 0, +0 / -0, +inf / -inf, NaN */
}
```

In practice, the interest of using the second piece of C code relies on the fact that it uses fewer tests, and the generated assembly code will contain fewer instructions. Hence, we expect that it will be faster.

Detecting whether x belongs to $\{\pm 0, \pm\infty, \text{NaN}\}$

To decide if $x \in \{\pm 0, \pm\infty, \text{NaN}\}$ we may use a similar method to the one presented above. However we can now restrict to the absolute value $|x|$ of x, whose integer encoding is $absX$ (see Section 2.1.2). Clearly, $x \in \{\pm 0, \pm\infty, \text{NaN}\}$ if and only if $|x| \in \{+0, +\infty, \text{NaN}\}$. Using k-bit integer arithmetic modulo 2^k, this is equivalent to:

$$absX - 1 \geq 2^{k-1} - 2^{p-1} - 1.$$

For $(k, p, e_{\max}) = (32, 24, 127)$, this condition can be implemented using the following piece of C code.

```
absX = X & 0x7FFFFFFF;
if((absX - 1) >= 0x7F7FFFFF){
  /* x < 0, +0 / -0, +inf / -inf, NaN */
}
```

3.4.3 How to determine the output to be returned?

Now let us consider that x is a special input. To decide which output has to be returned, let us consider two cases: n is positive or n is negative.

When n is positive

Fom Table 3.4, in this case, we observe that if x is a special input, we return

- x or a quiet NaN (qNaN) if $n = 2$ or n is odd,
- $|x|$ or a quiet NaN (qNaN) otherwise.

When $n = 2$. In this case, we have to return x when $x \in \{\pm 0, +\infty\}$ and qNaN otherwise. Consider that X is the k-bit unsigned interger encoding of x. Using Table 1.4, the output can be decided from the bit string of X as follows (since x is known to be a special input): we return X if

$$X \leq (2^{k-1} - 2^{p-1}) \ \vee \ X = 2^{k-1}.$$

If this condition is not satisfied, that means that x is either $-\infty$, NaN, or a negative input ($x < 0$), and a quiet NaN (qNaN) has to be returned. Since the IEEE 754-2008 standard does not specify

the sign of the NaN, the returned quiet NaN can be constructed by setting the bits $X_{k-2}, \ldots, X_{p-2}$ to 1 and leaving the others unchanged. This can be done by taking the bitwise OR between X and $2^{k-1} - 2^{p-2}$. The full special input handling can thus be implemented for $(k, p, e_{\max}) = (32, 24, 127)$, for example, using the following piece of C code.

```
if((X - 1) >= 0x7F7FFFFF){
  if((X <= 0x7F800000) || (X == 0x80000000)) return X;
  return X | 0x7FC00000;
}
```

When n is odd. In this case, the returned output is also x or qNaN. However, this output is easier to construct, since the only situation where it is a qNaN is when x is a NaN. Assuming that x is a NaN, the quiet NaN to be returned may be constructed by setting the bit X_{p-2} to 1, that is, by taking the bitwise OR between X and 2^{p-2}. Since from Table 1.4 x is NaN if and only if $absX > 2^{k-1} - 2^{p-1}$, an implementation for $(k, p, e_{\max}) = (32, 24, 127)$ is as follows:

```
if((absX - 1) >= 0x7F7FFFFF){
  if(absX > 0x7F800000) return X | 0x00400000;
  return X;
}
```

When n is even and $\neq 2$. In this last case, the output can be deduced in a similar way to what has been done when $n = 2$, except that $|x|$ is returned when $x \in \{\pm 0, +\infty\}$. The following implementation for $(k, p, e_{\max}) = (32, 24, 127)$ can be easily derived.

```
if((X - 1) >= 0x7F7FFFFF){
  if((X <= 0x7F800000) || (X == 0x80000000)) return absX;
  return X | 0x7FC00000;
}
```

When n is negative

From Table 3.5, in this second case, we deduce that if x is a special input then we return

- $+0$, $\pm\infty$, or qNaN if $n = -2$,
- ± 0, $\pm\infty$, or qNaN if is odd,
- $+0$, $+\infty$, or qNaN otherwise.

When $n = -2$. In this case, we have to return $+0$ when $x = +\infty$, ∞ when $x = 0$ (with the correct sign), and qNaN otherwise. Recall that x is known to be a special input. Assuming that X is the k-bit unsigned integer encoding of x, the output can also be decided from the bit string of X. Using Table 3.6, we can deduce that if $x = \pm 0$ or $x = +\infty$, then $X \leq 2^{k-1} - 2^{p-1}$ or $X = 2^{k-1}$, and the returned output may be obtained by taking the XOR between X and $2^{k-1} - 2^{p-1}$. Otherwise, x is either $-\infty$, NaN, or a negative input ($x < 0$), and since the IEEE 754-2008 standard does not specify the sign of the NaN's, the returned quiet NaN can be constructed by taking the bitwise OR between X and $2^{k-1} - 2^{p-2}$, as for $n = 2$. The full special input handling can thus be implemented for $(k, p, e_{\max}) = (32, 24, 127)$, for example, using the following piece of C code:

```
if((X - 1) >= 0x7F7FFFFF){
  if((X <= 0x7F800000) || (X == 0x80000000)) return X ^ 0x7F800000;
  return X | 0x7FC00000;
}
```

Input x	Integer X	Returned value bit string
-0	$1\underbrace{00\cdots00}_{w \text{ bits}}00\cdots00$	$1\underbrace{11\cdots11}_{w \text{ bits}}00\cdots00$
$+0$	$0\underbrace{00\cdots00}_{w \text{ bits}}00\cdots00$	$0\underbrace{11\cdots11}_{w \text{ bits}}00\cdots00$
$+\infty$	$0\underbrace{11\cdots11}_{w \text{ bits}}00\cdots00$	$0\underbrace{00\cdots00}_{w \text{ bits}}00\cdots00$

Table 3.6: Relationship between input x, encoding integer X, and the bit string of the returned value, for $x \in \{\pm 0, +\infty\}$.

When n is odd. In the second case, the returned output is a quiet NaN if and only if the input x is a NaN. This quiet NaN may be constructed as when $n > 0$ is odd, by taking the bitwise OR between X and 2^{p-2} that is by setting the bit X_{p-2} to 1. Otherwise, from Table 3.6, we observe that the output can be obtained by taking the bitwise XOR between X and $2^{k-1} - 2^{p-1}$. For $(k, p, e_{\max}) = (32, 24, 127)$, this special input handling can be implemented using the following piece of C code:

```
if((absX - 1) >= 0x7F7FFFFF ){
  if(absX > 0x7F800000) return X | 0x00400000;
  return (X ^ 0x7F800000);
}
```

When $n \neq -2$ is even. For this last case, the output to be returned may be deduced in a similar way as for $n = -2$, except for $x = -0$ for which the output is also $+\infty$ [IEE08]. Hence that relies on adjoining the correct sign to the returned output, which is 0 if $x = \pm 0$, and X_{k-1} otherwise. Adjoining the correct sign, especially when $x = -0$, may be done in several ways. Let $x = -0$ and $X = 2^{k-1}$. Following what is done for both previous cases, taking the XOR between X and $2^{k-1} - 2^{p-1}$ leads to $2^k - 2^{p-1}$, which encodes $-\infty$ (see Table 1.4). In such a case, adjoining the correct sign relies on adding 2^{k-1}. In this special case, the carry is propagated outside the k-bit unsigned integer (since additions are done module 2^k), and we get $2^{k-1} - 2^{p-1}$ which encodes $+\infty$ as required. For $(k, p, e_{\max}) = (32, 24, 127)$, this special input handling can be implemented using the following piece of C code:

```
if((X - 1) >= 0x7F7FFFFF){
  sign_of_zero = (X == 0x80000000) << 31;
  if((X <= 0x7F800000) || (X == 0x80000000))
    return ((X ^ 0x7F800000) + sign_of_zero);
  return X | 0x7FC00000;
}
```

General remark concerning quiet NaN output

When a qNaN has to be returned, we contruct the output integer using `X | 0x00400000;` or `X | 0x7FC00000;`. And in both cases we observe that the returned qNaN keeps as much of the information of X as possible, as recommended in [IEE08, §6.2]:

- the payload is preserved when quieting an signaling sNaN (sNaN),
- x is returned when x is a quiet NaN (qNaN).

3.5 First application example: square root ($n = 2$)

This section is devoted to the implementation of the square root in the *binary32* format, as presented in [JKMR08]. The objective is to show the interest on a first example of the uniform approach presented above. Thus let us now consider that $n = 2$. From Table 3.4, we know that the function $\sqrt{x} = x^{1/2}$ is defined for $x > 0$. Here, we do not discuss the handling of special input (when $x \in \{\pm 0, \pm\infty, \text{NaN}\}$), which has just been presented in Section 3.4. Hence assume in this section that x is a nonzero positive (sub)normal floating-point number.

The result being always positive, we do not present the computation of the sign of the result. Hence, first we recall the range reduction of the square root. Then we will see how to compute efficiently the exponent D of the result. Then, we will detail the way used for constructing the evaluation points ($\hat{s}^*$, t^*) and for approximating the function. Finally, we will explain how to implement the rounding condition.

3.5.1 Range reduction for square root

Recall now the range reduction we have seen in Section 3.1.2, for the particular case $n = 2$ and $\mu = 0$. Let r be the exact result of the function $x^{1/2}$. By definition, we know from (3.10) and (3.11) that the exact result $r = x^{1/2}$ is as follows

$$r = (2^{c/2} \cdot m'_x)^{1/2} \times 2^d,$$

with $c = e'_x \bmod 2 \in \{0, 1\}$ and $d = \lfloor e'_x/2 \rfloor = (e'_x - c)/2$. More particularly, we have

$$r = \begin{cases} (m'_x)^{1/2} \times 2^{e'_x/2} & \text{if } e'_x \text{ is even,} \\ 2^{1/2} \cdot (m'_x)^{1/2} \times 2^{(e'_x-1)/2} & \text{if } e'_x \text{ is odd.} \end{cases}$$

Here, we express the exact value ℓ as in Section 3.1.2 (for $n > 0$), that is, from (3.12) together with (3.13), we have

$$\ell = s \cdot (m'_x)^{1/2} \quad \text{with} \quad s \in \{1, 2^{1/2}\}.$$

By Property 3.3, the value ℓ lies in the range $[1, 2)$, and more particularly by Corollary 3.1, the exact value ℓ is strictly less than $2 - 2^{-p}$ which the middle between $2 - 2^{1-p}$ and its successor 2. Hence for $\circ = \text{RU}_p$, we have $\circ(\ell) \in [1, 2]$, otherwise $\circ(\ell) \in [1, 2)$.

Finally from Property 3.1, we know that the exact result r never denormalizes nor underflows and overflows. From these statements, we conclude that we shall return the correctly-rounded value $\circ(r)$ defined as in (3.5), that is,

$$\circ(r) = \circ(\ell) \cdot 2^{e_r}, \quad \text{with} \quad e_r = d \quad \text{and} \quad e_r \in \{e_{\min}, \ldots, e_{\max}\}, \tag{3.47}$$

It remains now to see how to implement the exponent d and the correctly-rounded significand $m_r = \circ(\ell)$.

3.5.2 Result exponent computation

In this section we explain how to implement the computation of the result exponent D defined in (2.21). Indeed as we have seen in Section 2.3, and since the exact result r is always positive ($s_r = 0$), the k-bit unsigned integer R encoding the correctly-rounded value $\circ(r)$ is defined as follows

$$R = D \cdot 2^{p-1} + M_r.$$

It remains now to compute D and M_r. In this section, we focus on the computation of the integer exponent D, while the computation of the integer significand M_r is discussed in Section 2.2.4 and uses what is done in Section 3.5.6.

From Section 3.3.2, we know that when $n \geq 2$, that is $|n| \geq 2$ and $\mu = 0$, the integer D is defined as in (3.40), for $\mu = 0$:

$$D = \lfloor (E_x - n_x - MX + C/n \rfloor \quad \text{with} \quad C = (1-n) \cdot e_{\min} + w.$$

Finally for $(k, p, e_{\max}) = (32, 24, 127)$ and $w = 8$, we obtain $C = 134$, and D can be computed as follows:

$$D = \lfloor (E_x - n_x - MX + 134)/2 \rfloor.$$

The value MX can be computed using the instructions `max` and `nlz` available on the ST231 (see Section 2.3 and Listing 2.1 for more details). Recall that on an integer architecture, taking the integer part of a division by 2 can be done using a right shift by one bit, and do not need to implement the integer division presented in Section 3.3.2. Thus for $(k, p, e_{\max}) = (32, 24, 127)$, the computation of the result exponent D can be implemented as follows.

```
nx = ( X >= 0x00800000 );          Ex = X >> 23;

MX = maxu( nlz( X ) , 8 );

D = (Ex - nx - MX + 134) >> 1;
```

(The implementation of the remainder c of the integer division is presented in the next section, discussing the computation of the evaluation points $(\hat{s}^*, t^*)$.)

3.5.3 Extraction of the evaluation point $(\hat{s}^*, t^*)$

In Section 3.2.1, we have seen that the evaluation point (s^*, t^*) may be not representable with k bits, and has to be rounded. Section 2.1 and Listing 2.1 explain how to extract the input t^*, as $t^* = m'_x - 1$. Here we focus on the implementation of the computation of $\hat{s}^*$, the rounded value of s^* defined (3.11), and more particularly the integer S encoding the input $\hat{s}^*$ such that

$$S = \hat{s}^* \cdot 2^{k-1}.$$

Then, we explain how to compute the value c, that is, useful for computing S.

Computation of S

We know by definition that $s^* \in \{1, 2^{1/2}\}$ and may be unrepresentable on precision k. So let $\hat{s}^*$ be the rounded to nearest value of s^* in precision k. From (3.11) and the definition of $\hat{s}^*$, it follows that

$$\hat{s}^* = \begin{cases} 1 & \text{if } c = 0, \\ \underbrace{1.01101010000010011110011001100111111100111011111\cdots_2}_{k \text{ bits}} & \text{if } c = 1. \end{cases} \tag{3.48}$$

The value of s^* could be selected according to the value of c. This is actually a manner to deduce s^* that can be easily automated. Here we present how to extract the value s^*, and more particularly the integer S, by using logical and integer operations. We deduce from (3.48) that

$$\hat{s}^* = 1 + c \cdot (\underbrace{0.01101010000010011110011001100111111100111011111\cdots_2}_{k \text{ bits}}).$$

Recall that S is the k-bit unsigned integer encoding the input point $\hat{s}^*$, such as $S = \hat{s}^* \cdot 2^{k-1}$. The bit string of the integer S is defined as follows:

$$S = \begin{cases} 1000\cdots00 & \text{if } c = 0, \\ \underbrace{101*\cdots**}_{k \text{ bits}} & \text{if } c = 1. \end{cases}$$

where $* \in \{0, 1\}$. Since the two most significant bits of S are 10, the right value for S can be chosen by taking the bitwise AND between $101*\cdots**_2$ and

$$2^{k-1} + 2^{k-2} - c = \begin{cases} 1100\cdots00 & \text{if } c = 0, \\ \underbrace{1011\cdots11}_{k \text{ bits}} & \text{if } c = 1. \end{cases}$$

It follows that the integer S can be computed as

$$S = 101*\cdots**_2 \ \wedge \ (2^{k-1} + 2^{k-2} - c).$$

When $k = 32$, the value $\hat{s}^*$ belongs to $\{1, 1.01101010000010011110011001101 00\}$, and the integer S is as follows

$$S = \begin{cases} 10000000000000000000000000000000_2 = \text{0x80000000}_{16} & \text{if } c = 0, \\ 10110101000001001111001100110100_2 = \text{0xb504f334}_{16} & \text{if } c = 1. \end{cases}$$

Hence, given the value c, we deduce that $S = \text{0xb504f334}_{16} \wedge \big(\text{0xc0000000}_{16} - c\big)$, and its computation on the ST231 processor can be implemented as follows.

```
S = 0xB504F334 & (0xC0000000 - c);
```

Note that the values that s can take are actually encoded into the program. Indeed for large values of n ($n = 100$, for example), that leads to an increase of the latency for determining input s, as well as an increase of the size of the generated assembly code. Here we aim at proposing a uniform approach, and in the cases we have studied, the fact that the different values of s are directly encoded into the C codes is not an issue, since we have just studied the implementation of the function $x \mapsto x^{1/n}$ for small values of n like 2, 3, or 4.

Now it simply remains to see how to compute the value of c.

Computation of c

Recall that the value c is defined as the remainder of the division $\lfloor e'_x/2 \rfloor$, that is, defined as $c = e'_x \bmod 2$ or

$$c = \begin{cases} 0 & \text{if } e'_x \text{ is even}, \\ 1 & \text{if } e'_x \text{ is odd}. \end{cases}$$

Recall that x is a nonzero positive (sub)normal floating-point number and X the k-bit unsigned integer encoding it. Using $e'_x = e_x - \lambda_x$, we have $e'_x = E_x + e_{\min} - n_x - MX - w$. By definition $e_{\min}$ is an even integer, so e'_x is even (and $c = 0$) if and only if $E_x - n_x - MX - w$ is even, that is, if and only if the last bit of $E_x - n_x - MX - w$ equals 0. Finally, we get that

$$c \quad \text{is the last bit of} \quad E_x - n_x - MX - w.$$

For $(k, p, e_{\max}) = (32, 24, 127)$ and $w = 8$, since w is even, $c = 0$ if and only if $E_x - n_x - MX$ is even. This computation can be implemented as follows.

```
c = (Ex - nx - MX) & 0x1;
```

We may remark that in other formats for which the exponent width w is odd (in *binary64* format, for example), the value of c is not the last bit of $E_x - n_x - MX$ but its negation. In this case, the computation of c can be implemented as: `c = 1 - ((Ex - nx - MX)& 0x1);` or derived form.

Delay on S compared to the trailing significand T_x

We deduced from Section 2.1.3 that the first evaluation t^* encoded into the k-bit unsigned integer T_x can be obtained in 3 cycles, assuming unbounded parallelism.

Here, with unbounded parallelism, the computation of c consists in computing MX together with $E_x - n_x$ (2 cycles since x is positive), then computing c (4 cycles). Finally, S is obtained in 2 more cycles, thus in 6 cycles, that is with a delay of 3 cycles compared to the first input T_x.

How to cope with this delay will be explained in Chapter 5 discussing the generation of certified and efficient evaluation program.

3.5.4 Bivariate approximant computation for *binary32* implementation

We have seen in introduction that the implementation of square root may be done through several methods. The new approach we proposed here is based on the efficient evaluation of a *single bivariate* polynomial P, defined in Section 3.2.1. Here we present the way used to compute this polynomial P and the evaluation program we have chosen for evaluating it.

Bivariate approximant computation

Assume now $(k, p, e_{\max}) = (32, 24, 127)$. As described in Section 3.2.1, the goal is now to compute a bivariate polynomial P defined as

$$P = 2^{-25} + s \cdot a(t), \text{ with } s \in \{1, 2^{1/2}\} \text{ and } t \in \mathcal{T} = [0, 1 - 2^{-23}].$$

It follows from (3.30) that here the polynomial $a(t)$ has to be a univariate polynomial that approximates the function $(1+t)^{1/2}$ over $\mathcal{T}$ with an approximation error $\alpha(a)$, defined in (3.30) for $n = 2$, as follows

$$\alpha(a) < (2^{-25} - 2^{-31.5})/2^{1/2} \approx 2^{-25.51}.$$

In order to compute such a polynomial $a(t)$, we use the Remez' algorithm [Rem34] of Sollya. Table 3.7 below shows an estimation of $\alpha(a^*)$, with a^* the *minimax* polynomial of degree δ (defined in Definition 3.3), when δ varies from 6 to 11. From Table 3.7, we deduce that the polynomial $a(t)$

δ	6	7	8	9	10	11
$\alpha(a^*)$	$2^{-22.47}$	$2^{-25.31}$	$2^{-28.12}$	$2^{-30.89}$	$2^{-33.64}$	$2^{-35.63}$

Table 3.7: Numerical estimation of $\alpha(a^*)$, for $6 \leq \delta \leq 11$.

has to be of degree at least 8 to satisfy the bound $(2^{-25} - 2^{-31.5})/2^{1/2} \approx 2^{-25.51}$. Since it has to be evaluated at runtime, we choose a polynomial of smallest degree, that is, $\delta = 8$. The same degree could have been deduced using the `guessdegree` function of Sollya, as described in Section 3.2.3 and Listing 3.1.

Once we have determined the degree δ of the polynomial approximant, it remains to compute a machine-representable polynomial $a(t)$, that is, in our case, a polynomial $a(t)$ having k-bit coefficients. Assuming $k = 32$, Table 3.8 shows the coefficients of the polynomial $a(t)$ obtained by truncating on 32 bits the coefficients of the Remez's polynomial approximant, so that $a(t) = \sum_{i=0}^{8} a_i \cdot t^i$ and

$$a_0 = A_0 \cdot 2^{-31} \quad \text{and} \quad a_i = (-1)^{i+1} A_i \cdot 2^{-31}, \text{ with } 1 \leq i \leq 8,$$

where the integer A_i encodes the coefficient a_i in absolute value in $Q_{1.31}$ format, defined in Section 1.2.3. Here the coefficients are stored in absolute value in the k-bit integers, so that we gain one bit of accuracy on each coefficient since we do not store the sign. Therefore while writing the evaluation program, we will have to handle these coefficient signs through the choice of appropriate arithmetic operators (see paragraph "Unsigned fixed-point evaluation and arithmetic operator choice" of Section 5.1.1 for details).

Coefficient	Sign	Value	Encoding integer	Format
a_0	+	1.00000000325962901115417480468 75	`0x80000007`	$Q_{1.31}$
a_1	+	0.49999940209090709686279296875	`0x3ffffafc`	$Q_{1.31}$
a_2	-	0.12498275609686970710754394531 25	`0x0fff6f59`	$Q_{1.31}$
a_3	+	0.06230625417083501815795898437 5	`0x07f9a6be`	$Q_{1.31}$
a_4	-	0.03794703539460897445678710937 5	`0x04db72ce`	$Q_{1.31}$
a_5	+	0.02358471788465976715087890625	`0x0304d2f4`	$Q_{1.31}$
a_6	-	0.01247844425961375236511230468 75	`0x0198e4c7`	$Q_{1.31}$
a_7	+	0.00451590353623032569885253906 25	`0x0093fa25`	$Q_{1.31}$
a_8	-	0.00078448653221130371093 75	`0x0019b4c0`	$Q_{1.31}$

Table 3.8: Coefficients of polynomial approximant for square root implementation.

Once the polynomial approximant is known, we compute the certified approximation error bound θ defined in (3.31), as described in Section 3.2.3 and Listing 3.1, that is, by using the supremum norm algorithm of Sollya (see [CL07] or [CJL09]). We conclude that the approximation error $\alpha(a)$ of the polynomial $a(t)$ with respect to $(1+t)^{1/2}$ over $\mathcal{T}$ is such as

$$\alpha \leq \theta \quad \text{with} \quad \theta = 16630643740039556739018598713784467 1 \times 2^{-145} \approx 2^{-27.99}. \tag{3.49}$$

We observe that the certified bound θ is strictly less than $(2^{-25} - 2^{-31.5})/2^{1/2} \approx 2^{-25.51}$ as required in (3.31).

Finally, let us determine the evaluation error bound η. Given this certified approximation error bound θ in (3.49), we have now to evaluate $P(s,t) = 2^{-25} + s \cdot a(t)$ with a finite-precision straight line evaluation program $\mathcal{P}$, such as $\rho(\mathcal{P}) \leq \eta$ and

$$\begin{aligned} \eta &= \texttt{RoundDownward}(2^{-25} - \theta \cdot 2^{1/2} - 2^{-31.5}) \\ &= 26983728580722749777378236282278411 7 \times 2^{-143} \approx 2^{-25.30}, \end{aligned} \tag{3.50}$$

as in (3.37).

Before seing how to evaluate the polynomial $P(s,t)$ in next section, let us first see how to compute a polynomial approximant optimized for integer processors, like the ST231.

Structured polynomial for evaluating the *binary32* square root

As we have seen in Chapter 1, multiplications by powers of two may lead to better performances (in terms of evaluation latency and code size). Hence, here we propose to construct a new polyno-

mial by favoring coefficients to be powers of two. This polynomial is called *structured coefficient polynomial*. Using it for implementing the square root may lead to latency as well as code size reductions. We may find the polynomial $a(t)$ having the most structured coefficients through an iterative process, by forcing each coefficient to be a power of two.

Thus in this case, we have found a degree-8 polynomial approximant $a(t)$ with 4 structured coefficients:

$$a_0 = 1, \quad a_1 = 2^{-1}, \quad a_2 = 2^{-3}, \quad \text{and} \quad a_8 = 2^{-10},$$

as shown in Table 3.9 below.

Coefficient	Sign	Value	Encoding integer	Format
a_0	+	1	`0x80000000`	$Q_{1.31}$
a_1	+	0.5	`0x40000000`	$Q_{1.31}$
a_2	-	0.125	`0x10000000`	$Q_{1.31}$
a_3	+	0.062456594780087471008300781 25	`0x07FE93E4`	$Q_{1.31}$
a_4	-	0.038542577996850013732910156 25	`0x04EEF694`	$Q_{1.31}$
a_5	+	0.024822981562465429306030273 4375	`0x032D6643`	$Q_{1.31}$
a_6	-	0.013879625592380762100219726 5625	`0x01C6CEBD`	$Q_{1.31}$
a_7	+	0.005332766566425561904907226 5625	`0x00AEBE7D`	$Q_{1.31}$
a_8	-	0.0009765625	`0x00200000`	$Q_{1.31}$

Table 3.9: Coefficients of polynomial approximant for square root implementation.

This polynomial approximant $a(t)$ approximates the function $(1+t)^{1/2}$ over $\mathcal{T}$ with an approximation error $\alpha(a)$ such that

$$\alpha \leq \theta \quad \text{with} \quad \theta = 67748905354568046998735930043171 1267 \times 2^{-145} \approx 2^{-25.97}. \tag{3.51}$$

As required in (3.31), the certified error bound θ is strictly less than $(2^{-25} - 2^{-31.5})/2^{1/2} \approx 2^{-25.51}$. However, we observe that this bound is slightly larger than the one in the previous case, and the polynomial has to be evaluated more accurately. Once the polynomial $a(t)$ and the bound θ are known, it remains to determine the evaluation error bound η, so that $\rho(\mathcal{P}) \leq \eta$. Using (3.37), it follows that

$$\eta = 17821387731344410818552897184130923 5 \times 2^{-144} \approx 2^{-26.89}. \tag{3.52}$$

In the remainder of this section, we will consider these two polynomial approximants. We will now see how to evaluate them on the ST231 processor.

3.5.5 Bivariate polynomial evaluation program

The problem of evaluating the function is finally reduced to the evaluation of the bivariate polynomial $P(s,t)$ defined in (3.24). Recall that we denote by $\mathcal{P}$ the evaluation program that evaluates this polynomial $P(s,t)$. The objective is thus to compute a k-bit unsigned integer V, defined as

$$V = v \cdot 2^{k-2} \quad \text{with} \quad v = \mathcal{P}(\hat{s}^*, t).$$

Let us now see how to evaluate P efficiently. By efficiently, we mean with an evaluation program that reduces the evaluation latency as well as the number of operations, especially the number of multiplications.

Before presenting the programs, recall that the target architecture is the ST231 processor core, which is a 4-issue VLIW 32-bits integer processor on which only 2 (pipelined) multiplications can

be launched each cycle. As we have already said in Section 3.2.4, several methods may be used for evaluating the polynomial P. These methods are presented in Chapter 5. And to be able to point out the efficiency of the polynomial evaluation programs proposed in the section, let us observe that the most classical evaluation program, Horner's rule would evaluate the polynomial P in $4 \cdot (\delta + 1)$ cycles, that is, 36 cycles for the square root. We expect that we can evaluate this polynomial faster, thus it turns out to be interesting to find a *best* evaluation program for evaluating P. These schemes have been found using CGPE described in Chapter 6. Here we present the results, while the method used to obtain them is presented in Chapter 6.

Unstructured polynomial approximant evaluation

Let us first consider the polynomial P defined in Table 3.8, that does not have any structured coefficients.

First, we may consider a best scheme for evaluating the polynomial $a(t)$ and compute the final result using a last Horner's iteration. Using this, P is evaluated as follows:

$$\begin{aligned} P(s,t) &= 2^{-25} + s \cdot \Big[\big((a_0 + t \cdot a_1) + t^2 \cdot (a_2 + t \cdot a_3)\big) + \\ &\quad (t^2 \cdot t^2) \cdot \Big(\big((a_4 + t \cdot a_5) + t^2 \cdot (a_6 + t \cdot a_7)\big) + (t^2 \cdot t^2) \cdot a_8\Big)\Big]. \end{aligned} \tag{3.53}$$

With this first *best scheme*, the polynomial $a(t)$ may be evaluated in 13 cycles and the whole evaluation is achieved in 17 cycles. The last multiplication by s is a bottleneck for evaluation. Hence, let us now see how to distribute this multiplication by s over the evaluation of the polynomial $a(t)$. We get finally the following evaluation scheme:

$$\begin{aligned} P(s,t) &= \Big[\Big(2^{-25} + \big(s \cdot (a_0 + t \cdot a_1)\big)\Big) + \Big((s \cdot t^2) \cdot \big((a_2 + t \cdot a_3) + t^2 \cdot a_4\big)\Big)\Big] + \\ &\quad \Big[\Big((t \cdot t^2) \cdot (s \cdot t^2)\Big) \cdot \Big((a_6 + t \cdot a_7) + \big(t^2 \cdot (a_8 + t \cdot a_9)\big)\Big)\Big], \end{aligned} \tag{3.54}$$

that enables to evaluate the polynomial P in 13 cycles, that is, 4 cycles faster than the previous scheme.

These schemes have been implemented on ST231, as well as the one presented in Chapter 5. Table 3.10 summarizes the results. For each evaluation scheme, it presents the latency (L), the number of instructions (N), the number of instructions per cycle (IPC), and the code size (CS). In

	L (cycles)	N	IPC	CS (bytes)	$\rho(\mathcal{P})$	
					MPFI	Gappa
Horner's rule	36	19	0.52	144	$2^{-28.0632}$	$2^{-28.0633}$
2nd order Horner's rule	23	21	0.91	152	$2^{-27.843}$	$2^{-27.843}$
Estrin's method	18	21	1.17	136	$2^{-28.0632}$	$2^{-28.0633}$
2nd Estrin's method	16	22	1.38	144	$2^{-27.8397}$	$2^{-27.8398}$
Best univariate	17	22	1.29	144	$2^{-28.0632}$	$2^{-28.0633}$
Best bivariate	13	23	1.77	152	$2^{-27.9362}$	$2^{-27.9362}$

Table 3.10: Performances on ST231, for square root implementation using unstructured polynomial coefficients.

this table, the latency L corresponds to the evaluation latency on unbounded parallelism as well as on a simplified model of the ST231 processor and in practice on the ST231. In each case, the evaluation program is scheduled *optimally* (in terms of latency) by the `st200cc` compiler.

We can observe that the best evaluation program found here (called *Best bivariate* in the table) is 2.77 times faster than Horner's rule. Remark also that it turns out to be efficient to distribute the multiplication by s over the evaluation of the polynomial $a(t)$. Otherwise, we would get an evaluation program almost as efficient as the Estrin's method.

The C code for the implementation of the square root using the best bivariate evaluation program is given in Listing 3.3. Here and hereafter, comment `// 1.31` in listings indicates that the corresponding integer encodes an unsigned fixed-point value in the format $Q_{1.31}$ (see line 3 of Listing 3.3 below, for example), while `// s0.31` indicates that the integer encodes a signed fixed-point value in the format $Q_{0.31}$ with one bit for handling the sign (see line 5 of Listing 5.1 (Chapter 5), for example). For this program, the IPC of 1.77 shows the parallel nature (as shown on the evaluation tree in Figure 3.2 as well) of the evaluation scheme compared to the others. This feasible schedule of this program on ST231 is presented in Table 3.11. Remark here that three issues are enough.

```
1  uint32_t __best_bivariate_eval__ (uint32_t x, uint32_t y)
2  {
3    uint32_t r0  = mul(x, 0x3ffffafc);      // 1.31
4    uint32_t r1  = 0x80000007 + r0;         // 1.31
5    uint32_t r2  = mul(y, r1);              // 2.30
6    uint32_t r3  = 0x00000020 + r2;         // 2.30
7    uint32_t r4  = mul(x, x);               // 0.32
8    uint32_t r5  = mul(y, r4);              // 1.31
9    uint32_t r6  = mul(x, 0x07f9a6be);      // 1.31
10   uint32_t r7  = 0x0fff6f59 - r6;         // 1.31
11   uint32_t r8  = mul(r4, 0x04db72ce);     // 1.31
12   uint32_t r9  = r7 + r8;                 // 1.31
13   uint32_t r10 = mul(r5, r9);             // 2.30
14   uint32_t r11 = r3 - r10;                // 2.30
15   uint32_t r12 = mul(x, r4);              // 0.32
16   uint32_t r13 = mul(r12, r5);            // 1.31
17   uint32_t r14 = mul(x, 0x0198e4c7);      // 1.31
18   uint32_t r15 = 0x0304d2f4 - r14;        // 1.31
19   uint32_t r16 = mul(x, 0x0019b4c0);      // 1.31
20   uint32_t r17 = 0x0093fa25 - r16;        // 1.31
21   uint32_t r18 = mul(r4, r17);            // 1.31
22   uint32_t r19 = r15 + r18;               // 1.31
23   uint32_t r20 = mul(r13, r19);           // 2.30
24   uint32_t r21 = r11 + r20;               // 2.30
25   return r21;
26 }
```

Listing 3.3: Best evaluation program for *binary32* square root implementation.

Evaluation program validation

Once we have written an efficient evaluation program (in Listing 3.3), we have to check if the evaluation entailed is less than the required bound $\eta \approx 2^{-25.30}$ in (3.50). This is done using Gappa [Mel], [Mel06], and presented in Section 5.1.1 (paragraph "Evaluation program validation"). The main objectives are to check:

- if each evaluation variable r_i fits in a 32-bit unsigned integer,
- and if it is accurate enough, that is, if its evaluation error is no larger than η in (3.50).

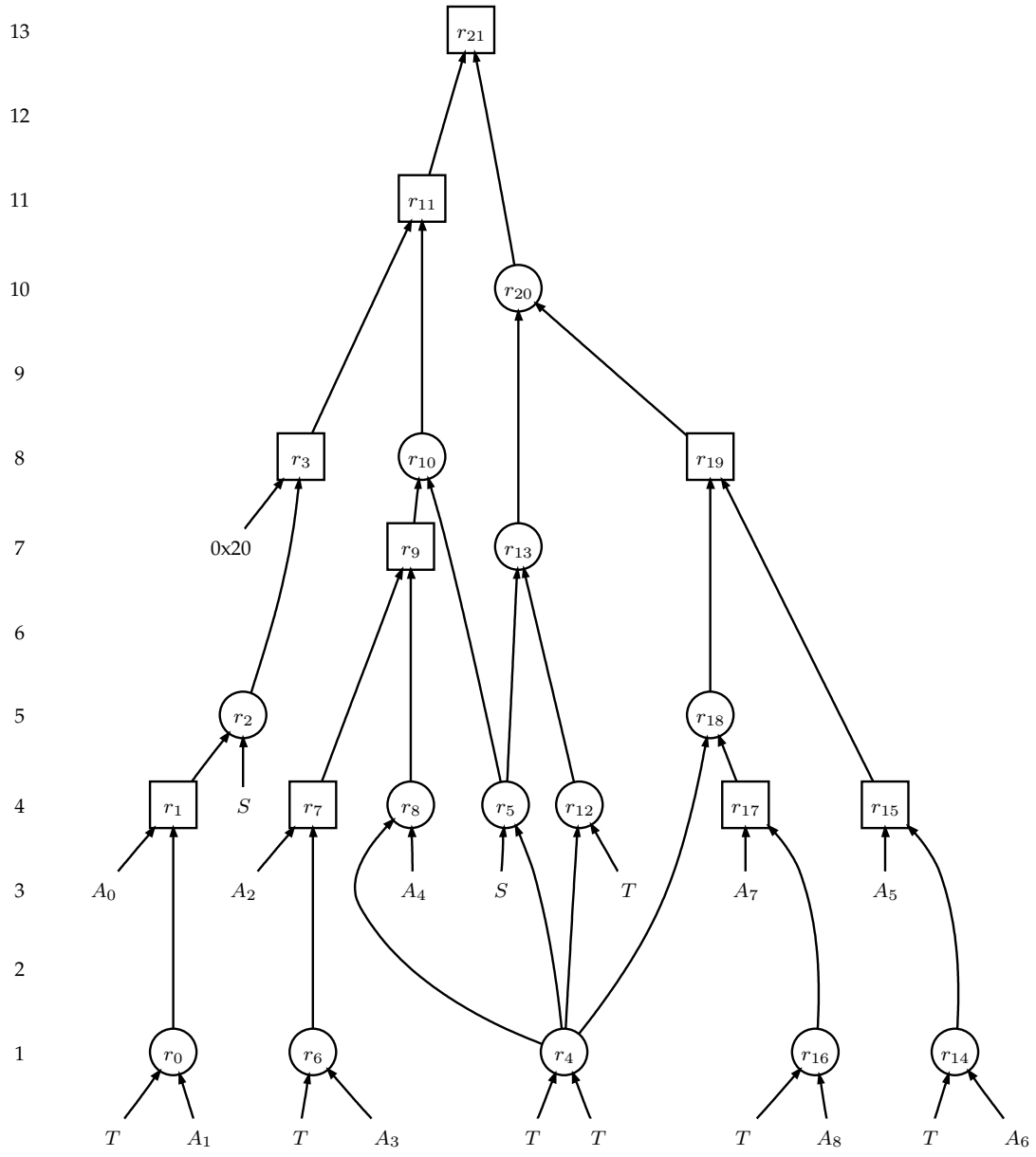


Figure 3.2: Evaluation tree for square root implementation using *best* scheme.

	Issue 1	Issue 2	Issue 3	Issue 4
Cycle 0	r_0	r_4		
Cycle 1	r_6	r_{16}		
Cycle 2	r_{14}			
Cycle 3	r_5	r_{12}		
Cycle 4	r_8	r_{17}		
Cycle 5	r_7	r_{18}		
Cycle 6	r_1	r_{13}		
Cycle 7	r_2	r_9	r_{15}	
Cycle 8	r_{10}	r_{19}		
Cycle 9	r_{20}			
Cycle 10	r_3			
Cycle 11	r_{11}			
Cycle 12	r_{21}			

Table 3.11: Feasible scheduling on ST231, for square root implementation.

Concerning the computation of the evaluation error bound, it has also been implemented using interval arithmetic with MPFI [CLN+] (see also [RR05]). This is mainly done by computing a certified enclosure of the error entailed by the evaluation of each instruction of the program. More precisely, given an instruction, it consists in propagating the errors entailed by the computation of its operands and then computing an interval enclosing the error on the considered instruction, where bounds are multiple-precision floating-point numbers. The last column of Table 3.10 shows the certified evaluation error computed using MPFI and Gappa.

We can observe that all the schemes implemented above are all accurate enough, and their evaluation errors are less than the certified bound η in (3.50). More example, using MPFI and Gappa, we may observe that the *best bivariate* generated evaluation program has an evaluation error less than the required bound $\eta \approx 2^{-25.30}$ in (3.50). Remark also that the evaluation error bounds, computed using both methods (MPFI or Gappa), are almost equivalent. In practice, we observe that the bound is faster to be computed using MPFI than Gappa.

Structured polynomial for evaluating the *binary32* square root

Let us now consider the structured polynomial approximant, whose coefficients are displayed in Table 3.9. As for the previous polynomial approximant, we have also implemented several evaluation methods. Table 3.12 shows, for each of these methods, the latency (L), the number of instructions (N), the number of instructions per cycle (IPC) and the code size (CS) of several evaluation schemes, while the last column shows the certified evaluation error bound η computed using interval arithmetic (MPFI) and Gappa.

Remark first, that all these evaluation programs are scheduled *optimally* (in term of latency) using the `st200cc` compiler. Concerning the impact of using structured coefficients, we observe that it may lead to a speed-up of up to 2 cycles, for "naive" method. But for the most efficient evaluation programs, we do not remark any speed-up. Actually these schemes expose a lot of instruction-level parallelism, and the gain due to structured coefficient is hidden by instruction-level parallelism exposure.

Furthermore, using structured coefficients leads to a code size reduction of up to 16 bytes, that is, a reduction of about 10.5 %, especially for the *best* generated program.

Therefore, the best evaluation scheme using *structured polynomial* has been kept for implementing square root in FLIP. Its evaluation error is strictly less than the required bound $\eta \approx 2^{-26.89}$

	L (cycles)	N	IPC	CS (bytes)	$\rho(\mathcal{P})$	
					MPFI	Gappa
Horner's rule	34	19	0.56	136	$2^{-28.0632}$	$2^{-28.0633}$
2^{nd} order Horner's rule	21	21	1	148	$2^{-27.842}$	$2^{-27.842}$
Estrin's method	17	21	1.24	128	$2^{-28.0632}$	$2^{-28.3578}$
2^{nd} Estrin's method	15	22	1.47	128	$2^{-27.8386}$	$2^{-28.0096}$
Best univariate	17	22	1.29	128	$2^{-28.0632}$	$2^{-28.3578}$
Best bivariate	13	23	1.77	136	$2^{-27.9347}$	$2^{-28.2016}$

Table 3.12: Performances on ST231, for square root implementation using *structured* polynomial coefficients.

defined in (3.52). Listing 3.4 displays the corresponding C code. We observe the multiplication by powers of two (A_1, A_2, and A_8) are implemented as simple shifts. Finally, Table 3.13 below displayed a feasible scheduling in 13 cycles of the program in Listing 3.4 below on ST231. Notice again that three of the four available issues are enough.

	Issue 1	Issue 2	Issue 3	Issue 4
Cycle 0	r_0	r_4	r_{14}	
Cycle 1	r_1	r_8		
Cycle 2	r_{12}			
Cycle 3	r_5	r_{15}	r_{16}	
Cycle 4	r_2	r_{17}		
Cycle 5	r_{13}	r_{18}		
Cycle 6	r_6	r_9	r_{11}	
Cycle 7	r_3			
Cycle 8	r_7	r_{19}		
Cycle 9	r_{10}	r_{20}		
Cycle 10				
Cycle 11				
Cycle 12	r_{21}			

Table 3.13: Feasible scheduling on ST231, for square root implementation with structured coefficient polynomial.

In Table 3.12 above, we can observe that the bounds computed using Gappa are sometimes tighter than the ones obtained using MPFI. This difference is due to the fact that MPFI is used for computing these bounds by "naive" interval arithmetic, while Gappa uses some rewriting rules and theorems in addition to "naive" interval arithmetic that may lead to tighter bounds (slightly tighter in our case).

3.5.6 How to implement the rounding condition?

Once we have the evaluation program $\mathcal{P}$ that computes the value v, we may write the full code for implementing the square root. However, it now remains to described how to implement the rounding condition, and more particularly the condition `cond` used in Section 2.2.4. Recall that the objective is to compute $\circ(\ell)$ from the value $v = \mathcal{P}(\hat{s}^*, t)$ that approximates the exact value ℓ from above such as $-2^{-p} \leq \ell - v < 0$. Assume from now that the value v is encoded in a k-bit unsigned integer V such as $V = v \cdot 2^{k-2}$. To implement the rounding procedure, recall that the value u

```
uint32_t __best_bivariate_eval_structured__(uint32_t T, uint32_t S)
{
  uint32_t r0  = T >> 2;                          // 1.31
  uint32_t r1  = 0x80000000 + r0;                 // 1.31
  uint32_t r2  = mul(S, r1);                      // 2.30
  uint32_t r3  = 0x00000020 + r2;                 // 2.30
  uint32_t r4  = mul(T, T);                       // 0.32
  uint32_t r5  = mul(S, r4);                      // 1.31
  uint32_t r6  = mul(T, 0x07fe93e4);              // 1.31
  uint32_t r7  = 0x10000000 - r6;                 // 1.31
  uint32_t r8  = mul(r4, 0x04eef694);             // 1.31
  uint32_t r9  = r7 + r8;                         // 1.31
  uint32_t r10 = mul(r5, r9);                     // 2.30
  uint32_t r11 = r3 - r10;                        // 2.30
  uint32_t r12 = mul(T, r4);                      // 0.32
  uint32_t r13 = mul(r12, r5);                    // 1.31
  uint32_t r14 = mul(T, 0x01c6cebd);              // 1.31
  uint32_t r15 = 0x032d6643 - r14;                // 1.31
  uint32_t r16 = T >> 11;                         // 1.31
  uint32_t r17 = 0x00aebe7d - r16;                // 1.31
  uint32_t r18 = mul(r4, r17);                    // 1.31
  uint32_t r19 = r15 + r18;                       // 1.31
  uint32_t r20 = mul(r13, r19);                   // 2.30
  uint32_t r21 = r11 + r20;                       // 2.30
  return r21;
}
```

Listing 3.4: Evaluation program for *binary32* square root implementation using *structured* coefficients.

denotes the value v truncated after p fraction bits:

$$u = u_{-1}u_0.u_1u_2\cdots u_p \quad \text{and} \quad 0 \le v - u < 2^{-p},$$

and by definition

$$-2^{-p} < \ell - u < 2^{-p},$$

as in (2.15) with $\lambda_r = 0$ (since the result never denormalizes).

Since the result of the square root is positive, the RoundTowardNegative is equivalent to the RoundTowardZero. Thus, we consider here that $\circ \in \{\mathrm{RN}_p, \mathrm{RU}_p, \mathrm{RD}_p\}$. As we have seen in Section 2.2.4, for the first two rounding-direction attributes (RN_p and RU_p), the rounding procedure relies on the test $u \ge \ell$, while for $\circ = \mathrm{RD}_p$ the procedure is based on the test $u > \ell$. Of course the value ℓ is not known exactly, and the rounding tests $u \ge \ell$ or $u > \ell$ cannot be implemented directly. However, it turns out that these can be implemented exactly by considering u^2 and ℓ^2 instead of u and ℓ.

To do so, let U be the k-bit unsigned integer encoding of u such as $U = u \cdot 2^{k-2}$, whose bit string is

$$U = 01v_1v_2\cdots v_p\underbrace{00\cdots00}_{k-p-2 \text{ bits}}, \tag{3.55}$$

and Q the k-bit unsigned integer encoding ℓ^2. By definition, we have $U = \lfloor V/2^{k-p-2} \rfloor \cdot 2^{k-p-2}$. It follows that the integer U may be computed by removing the $k-p-2$ last bits of the bit string of V, by taking the bitwise AND between V and $2^k - 2^{k-p-2}$. For $(k, p, e_{\max}) = (32, 24, 127)$, the following piece of C code shows how to compute U from V.

```
U = V & 0xFFFFFFC0;
```

Then, let P be the k-bit unsigned integer defined as

$$P = \texttt{mul}(U, U).$$

By definition of U and `mul`, we have the following

$$u^2 \cdot 2^{k-4} - 1 < P \leq u^2 \cdot 2^{k-4}. \tag{3.56}$$

From (3.11), recall that $\ell = 2^{c/2} \cdot (m'_x)^{1/2}$, and thus $\ell^2 = 2^c \cdot m'_x$. In fact, since $m'_x = m_x \cdot 2^{\lambda_x}$ with $\lambda_x \in \{0, \ldots, p-1\}$, we know that the exact value m'_x is representable with at most p bits. Hence, the value ℓ^2 is defined as follows:

$$\ell^2 = \begin{cases} m'_x = 1.m'_{1+\lambda_x} m'_{2+\lambda_x} \ldots m'_{p-1+\lambda_x} & \text{if } c = 0, \\ 2m'_x = 1m'_{1+\lambda_x}.m'_{2+\lambda_x} \ldots m'_{p-1+\lambda_x} & \text{if } c = 1, \end{cases}$$

and in both cases is representable with p bits. Here, we consider that Q is the k-bit unsigned integer defined such as

$$Q = \ell^2 \cdot 2^{k-4}. \tag{3.57}$$

Since $T_x = t_x \cdot 2^k$ and $m'_x = 1 + t_x$, we finally have

$$Q = 2^{c-3} \cdot (T_x/2 + 2^{k-1}).$$

We can remark here that this form for Q is preferred to $Q = 2^{c-4} \cdot (T_x + 2^k)$, since using k-bit integer, we would not have been able to compute $T_x + 2^k$. For $(k, p, e_{\max}) = (32, 24, 127)$, the implementation of the variable Q can be done as follows.

```
Q = ((Tx >> 1) + 0x80000000) >> (3 - c);
```

Finally it remains to explain how to implement exactly the rounding test $u \geq \ell$ (for $\circ \in \{\mathrm{RN}_p, \mathrm{RU}_p\}$) and $u > \ell$ (for $\circ = \mathrm{RD}_p$) using the integers P, Q, and U defined above.

Implementation of $u \geq \ell$, for RoundTiesToEven and RoundTowardPositive

Property 3.9. *The rounding test $u \geq \ell$ is equivalent to $P \geq Q$.*

Proof. By definition, since $u > 0$ and $\ell > 0$, we know that $u \geq \ell$ is equivalent to $u^2 \geq \ell^2$. Since $2^{k-4} > 0$, then $u^2 \geq \ell^2$ is equivalent to $u^2 \cdot 2^{k-4} \geq \ell^2 \cdot 2^{k-4}$. And from (3.57) and the left inequality in (3.56), we deduce that if $u^2 \geq \ell^2$ then $P + 1 > Q$ and $P \geq Q$ since P and Q are integers.

On the other hand, if $P \geq Q$ then $P \cdot 2^{4-k} \geq Q \cdot 2^{4-k}$. And from the right inequality in (3.56), we conclude that $u^2 \geq \ell^2$ and thus $u \geq \ell$. □

From Property 3.9, the condition $u \geq \ell$ is true if and only if $P \geq Q$, that is, if and only if the C condition `P >= Q` is satisfied. For $(k, p, e_{\max}) = (32, 24, 127)$, the rounding procedure for RoundToNearestTies and RoundTowardPositive can be implemented using the following piece of C code.

```
cond = (P >= Q);
```

Implementation of $u > \ell$, for RoundTowardNegative

In RoundTowardNegative, the rounding procedure is based on the test $u > \ell$ instead of $u \geq \ell$. But this condition cannot be implemented as directly as the previous one. To handle the case $u = \ell$, let us define $Q' \in \{0, 1\}$ such as:

$$Q' = \begin{cases} 1 & \text{if and only if } u^2 \cdot 2^{k-4} = P, \\ 0 & \text{otherwise.} \end{cases}$$

that is $Q' = 1$ if and only if $P = U^2 \cdot 2^{-k}$.

Property 3.10. *The rounding test $u > \ell$ is equivalent to $P \geq Q + Q'$.*

Proof. By definition, since $u > 0$ and $\ell > 0$, we know that $u > \ell$ is equivalent to $u^2 > \ell^2$. Since $2^{k-4} > 0$, then $u^2 > \ell^2$ is equivalent to $u^2 \cdot 2^{k-4} > \ell^2 \cdot 2^{k-4}$. Assume first that $Q' = 1$. From (3.57), (3.56), and the definition of Q', we deduce that the condition $u^2 > \ell^2$ is equivalent to $P > Q$ that is $P \geq Q + 1 = Q + Q'$.

Assume now that $Q' = 0$. Here we have $P \neq u^2 \cdot 2^{k-4}$, thus from (3.56) $P < u^2 \cdot 2^{k-4}$ and $P \geq Q$ implies $u^2 > Q \cdot 2^{4-k} = \ell^2$. On the other hand, if $u^2 > Q \cdot 2^{4-k}$ then from (3.56), we conclude that $P + 1 > Q$ and thus $P \geq Q = Q + Q'$. □

By definition, $1 \leq \ell < 2$ and $2^{k-4} \leq Q < 2^{k-2}$. We can deduce that for $Q + Q' < 2^k$ that is $Q + Q'$ fits exactly in a k-bit integer. Hence from Property 3.10, the condition $u > \ell$ is true if and only if $P \geq Q + Q'$ is true, that is, if and only the C condition `P >= Q + Qprime` is satisfied.

It remains now to compute the value Q'. In fact $Q' = 1$ if and only if $P = U^2 \cdot 2^{-k}$, that is $U^2 \cdot 2^{-k} = \texttt{mul}(U, U)$ and $U^2 \cdot 2^{-k} = U^2$. Assuming that k is even (which is the case for all the binary floating-point format presented in the IEEE 754-2008 standard), then $Q' = 1$ if and only if U is a multiple of $2^{-k/2}$. This can be deduced from the bit string of U in (3.55), and more particularly from its $k/2$ last bits. It can be easily done by taking a bitwise AND between U and $2^{k/2} - 1$, such as:

$$Q' = \begin{cases} 1 & \text{if and only if } \big(U \wedge (2^{k/2} - 1)\big) = 0, \\ 0 & \text{otherwise.} \end{cases}$$

Finally the implementation of this rounding procedure for $(k, p, e_{\max}) = (32, 24, 127)$ can be done as follows.

```
Qprime = ((U & 0x0000FFFF) == 0);

cond = (P >= Q + Qprime);
```

3.6 Second application example: reciprocal ($n = -1$)

In Section 3.5, we have presented some key elements for the efficient implementation of the square root. The second example we have chosen to illustrate the advantage of our uniform approach is the reciprocal ($n = -1$), since it corresponds to a special case when $n < 0$.

After some reminder on the range reduction for the reciprocal function, we will see how to compute the sign and the exponent of the result. Then, we will explain how to handle efficiently a given special input. Finally, we define the polynomial approximant and the methods used for ensuring the correct rounding.

3.6.1 Range reduction for the reciprocal

Recall here the range reduction described in Section 3.1.2 (for $n < 0$) for the particular case $n = -1$. As we have seen above the reciprocal is a special case of the function $x^{1/n}$. Assuming that x is a nonzero (sub)normal floating-point number, the exact result r for reciprocal is defined as $r = (-1)^{s_x} \times \ell \times 2^d$, with $\ell = 2 \cdot (m'_x)^{-1}$ and $d = -e'_x - 1$. And from Property 3.1, we know that the exact result r and its correctly-rounded value $\circ(r)$ may denormalize. Thus, in this case, we shall return the correctly-rounded result $\circ(r)$, defined as follows

$$\circ(r) = (-1)^{s_r} \cdot (\ell \cdot 2^{-\lambda_r}) \cdot 2^{e_r},$$

with

$$\lambda_r = \max(0, e_{\min} - d) \quad \text{and} \quad e_r = \max(e_{\min}, d).$$

From (3.7) and (3.8), we know that

$$\lambda_r \in \{0, 1, 2\} \quad \text{and} \quad d \in \{e_{\min} - 2, \cdots, e_{\max} - 2 + p\}.$$

Using this form, we ensure that the result exponent e_r is at least $e_{\min}$.

3.6.2 Result sign and exponent computation

In this section, we describe how to implement the computation of the sign and the exponent of the result from the input x, and more particularly from X the k-bit unsigned integer encoding x. Recall that in (2.21), the returned k-bit unsigned integer R that encodes the correctly-rounded result $\circ(r)$ is defined as follows

$$R = S_r + D \cdot 2^{p-1} + M_r.$$

From Section 3.3.1, we know that the computation of the sign s_r of the result is trivial since $s_r = s_x$, and $S_r = s_r \cdot 2^{k-1}$ can be obtained by taking a bitwise AND between X and 2^{k-1}.

Concerning the computation of the integer D, from Section 3.3.2, we know that when $n = -1$, the integer D is defined as follows:

$$D = \max(0, -(E_x - n_x - MX + C)),$$

with $C = 2e_{\min} + w + 1$. For $(k, p, e_{\max}) = (32, 24, 127)$ and $w = 8$ (which correspond to the floating-point parameters of the *binary32* floating-point format), we deduce that $C = -243$. Recall that MX can be computed using the instructions `max` and `nlz` available on ST231 (see Section 2.3 and Listing 2.1 for details).

For $(k, p, e_{\max}) = (32, 24, 127)$, the computation of the result sign S_r and exponent D can be implemented as in lines 1 and 9, respectively, of the following piece of C code.

```
1  absX = X & 0x7FFFFFFF;              Sr = X & 0x80000000;
2
3  nx = (absX >= 0x00800000);
4
5  Ex = absX >> 23;                    MX = maxu(nlz(absX) , 8);
6
7  tmp = (nx + MX - Ex - 243);
8
9  D = max (0 , tmp);                  Lr = max(0 , 0-tmp);
```

Here, the unsigned integer L_r encoding the value λ_r is computed in the same way as for the multplication in Section 2.4.3, and its computation is not detailed.

3.6.3 Special input, underflow, and overflow

From Property 3.1, we know that the the correctly-rounded $\circ(r)$ may denormalize, and in this cases, a special output has to be returned. Here we present an efficient way to handle overflow and special input in the same time.

Special input handling

First consider the special input handling. Let x be a floating-point datum. From Table 3.5, we observe that x is a special input if it is a special datum (± 0, $\pm\infty$, NaN), and is these cases, the IEEE 754-2008 standard [IEE08] specifies that a special value to be returned. To handle such input, we proceed as described in Section 3.4.2 for deciding if $x \in \{\pm 0, \pm\infty, \text{NaN}\}$. Let X and $absX$ be the k-bit unsigned integers encoding of the floating-point datum x, and its absolute value $|x|$: x is a special input if and only if

$$absX - 1 \geq 2^{k-1} - 2^{p-1} - 1. \tag{3.58}$$

Overflow characterization

We know from Property 3.1 that the exact result r may denormalize, and in this case, a special output has also to be returned. When the exact result overflows, that is, $|r| > \Omega$ then the following output shall be returned

$$(-1)^{s_r} \cdot \Omega \quad \text{or} \quad (-1)^{s_r} \cdot \infty,$$

according to the active rounding-direction, as described in Section 2.3.

Let us recall that x is a nonzero (sub)normal floating-point number. Assume now that the exact result $|r| = |1/x|$ is larger than Ω. From (3.7) and (3.8), and since in this case $\lambda_r = 0$ (the exact is not in the range of subnormal floating-point numbers), it follows that $|r| > \Omega$ if and only if $d > e_{\max}$, that is, using $e'_x = e_x - \lambda_x$

$$-e_x + \lambda_x - 1 > e_{\max}. \tag{3.59}$$

By definition of x in (1.3), it follows that $e_x \in \{e_{\min}, , \ldots, e_{\max}\}$. Hence, using $e_{\max} = 1 - e_{\min}$, we get

$$e_{\min} - 2 + \lambda_x \leq -e_x + \lambda_x - 1 \leq e_{\max} - 2 + \lambda_x,$$

and (3.59) holds if and only if

$$\lambda_x > 2 \quad \text{and} \quad |x| \leq 2^{e_{\min}-3}.$$

Recall that in this case L_x is defined in Section 2.1 as the number of leading zeros in the bit string of $absX$ (encoding $|x|$), as follows:

$$L_x = \lambda_x + w.$$

Hence we conclude that $|r| > \Omega$ if and only if $L_x \geq 3 + w$, that is the bit string of $absX$ has at least $3 + w$ leading zeros. Therefore, we deduce that $|r| > \Omega$ if and only if $0 < absX < 2^{k-w-3}$. And since addition and subtraction are done modulo 2^k, this condition can be rewrite as follows:

$$absX - 2^{k-w-3} > 2^k - 2^{k-w-3}. \tag{3.60}$$

If this condition is satisfied, it remains to decide which output has to be returned.

How detect special input and overflow together?

From (3.60), we know that x is a special value if and only if $absX - 1 \geq 2^{k-1} - 2^{p-1} - 1$. Since $2^{k-w-3} < 2^{k-1} - 2^{p-1} - 1$, then we deduce also that x is a special input if $absX - 2^{k-w-3} \geq 2^{k-1} - 2^{p-1} - 2^{k-w-3}$. From this statement, we deduce that we can detect special input as well as overflow in the same time. And finally, a special output shall be returned if

$$absX - 2^{k-w-3} \geq 2^{k-1} - 2^{p-1} - 2^{k-w-3}$$

as described in Table 3.14 below.

Input x	± 0	$x \leq 2^{e_{\min}-3}$	$x \geq -2^{e_{\min}-3}$	$\pm\infty$	NaN
$\mathrm{RN}_p(r)$		$\pm\infty$		± 0	qNaN
$\mathrm{RU}_p(r)$	$\pm\infty$	$+\infty$	$-\Omega$	± 0	qNaN
$\mathrm{RD}_p(r)$	$\pm\infty$	$+\Omega$	$-\infty$	± 0	qNaN
$\mathrm{RZ}_p(r)$	$\pm\infty$	$+\Omega$	$-\Omega$	± 0	qNaN

Table 3.14: Special output for x^{-1}.

From Table 3.14 and for $(k, p, e_{\max}) = (32, 24, 127)$, we obtain the following piece of C code for each rounding-direction attribute.

RoundTiesEven.

```
if((absX - 0x00200000) >= 0x7F600000){
  if(absX > 0x7F800000) return X | 0x00400000;
  return Sx | ((absX & 0x7F800000) ^ 0x7F800000);
}
```

RoundTowardPositive.

```
if((absX - 0x00200000) >= 0x7F600000){
  if(absX > 0x7F800000)  return X | 0x00400000;
  if(absX == 0x7F800000) return Sx;
  return (Sx | 0x7F800000) - ((Sx >> 31) & (absX != 0x00000000));
}
```

RoundTowardNegative.

```
if((absX - 0x00200000) >= 0x7F600000){
  if(absX >  0x7F800000) return X | 0x00400000;
  if(absX == 0x7F800000) return Sx;
  return (Sx | 0x7F7FFFFF) + ((Sx >> 31) & (absX == 0x00000000));
}
```

RoundTowardZero.

```
if((absX - 0x00200000) >= 0x7F600000){
  if(absX  > 0x7F800000) return X | 0x00400000;
  if(absX == 0x7F800000) return Sx;
  return (Sx | 0x7F800000) - (absX != 0x00000000);
}
```

Special case implementation

The last case to be handled is the so-called *special case implementation* defined in Section 2.3.1. Indeed this case may occur when $\ell = 2$ and $e_{\max}$, and $r = \pm 2^{e_{\max}}$. From Section 2.3.1, we know that this case occurs:

- in RoundTowardPositive: when $\ell = 2$, $d = e_{\max}$ and $s_r = 1$,
- in RoundTowardNegative: when $\ell = 2$, $d = e_{\max}$ and $s_r = 0$,
- in RoundTowardZero: when $\ell = 2$, $d = e_{\max}$,

and in all these situations, $\pm\Omega$ shall be returned (see 2.3.1 for more details). Notice that these special cases never occur in RoundTiesToEven mode. By definition of ℓ in (3.11), we observe that $\ell = 2$ if and only if $m'_x = 1$, and since $d = -e'_x - 1$, $d = e_{\max}$ means that $e'_x = e_{\min} - 2$. We conclude that *special implementation cases* must be handled if

$$x = (-1)^{s_x} \cdot 2^{e_{\min}-2} \quad \text{and} \quad X = S_x + 2^{p-3},$$

according to the sign of the result and the active rounding-direction attribute, as presented below.

For each rounding-direction attribute for which these cases may occur, this special input may be handled separately has described for $(k, p, e_{\max}) = (32, 24, 127)$ in the C codes below.

RoundTowardPositive: if $X = 2^{k-1} + 2^{p-3}$ then return $-\Omega$

```
if(X == 0x80200000) return 0xFF7FFFFF;
```

RoundTowardNegative: if $X = 2^{p-3}$ then return $+\Omega$

```
if(X == 0x00200000) return 0x7F7FFFFF;
```

RoundTowardZero: if $absX = 2^{p-3}$ then return $(-1)^s_x \cdot \Omega$

```
if(absX == 0x00200000) return (Sx | 0x7F7FFFFF);
```

3.6.4 How to approximate the exact value ℓ ?

Until now we have seen how to compute the sign and the exponent of the result, and how to handle overflow together with special input. It remains now to study how to approximate the exact value ℓ in (3.11). This section presents now how to build the polynomial P defined in (3.24) and how to write an evaluation program $\mathcal{P}$ so that the condition (3.29) holds.

Univariate polynomial approximation

This uniform approach consists in approximating the function to be evaluated by a single bivariate polynomial P defined in (3.24), and evaluating it by an efficient evaluation program $\mathcal{P}$, such as the overall error stays strictly less than 2^{-p-1} and (3.29) holds. Recall that the function $F(s,t) = 2^{-p-1} + s \cdot 2^{-1/n} \cdot (1+t)^{1/n}$ has to be evaluated at the input (s^*, t^*) such that $s^* = 2^{c/n}$ and $t^* \in \mathcal{T} = m'_x - 1$. By definition when $n = -1$ we know that $c = 0$, and we deduce that $s^* = 1$. Remark from now that the input extraction is done in a simpler way than for square root. Actually, the value s^* will not be stored and the value t^* will be computed as presented in Section 2.1 and

Listing 2.1. And the approximation v of ℓ, and its integer encoding $V = v \cdot 2^{k-2}$, will be thus computed as

$$v = \mathcal{P}(t^*).$$

Therefore, the case when $n = -1$ can be reduced to the evaluation of the function

$$F(t) = 2^{-p-1} + 2/(1+t).$$

As usual, the function $2/(1+t)$ is approximated over $\mathcal{T}$ by a univariate polynomial $a(t)$. From (3.30) and since $\gamma(s) = 0$ when $n = -1$, for $(k, p, e_{\max}) = (32, 24, 127)$ the approximation error of the polynomial $a(t)$ with respect to the function $2/(1+t)$ over $\mathcal{T}$ has to be strictly less than 2^{-25}. Table 3.15 shows an estimation of $\alpha(a^*)$, with a^* the *minimax* polynomial of degree δ (defined in Definition 3.3), while δ varies from 6 to 11. From Table 3.15, we deduce that the imple-

δ	6	7	8	9	10	11
$\alpha(a^*)$	$2^{-16.25}$	$2^{-18.80}$	$2^{-21.34}$	$2^{-23.88}$	$2^{-26.43}$	$2^{-28.97}$

Table 3.15: Numerical estimation of $\alpha(a^*)$, for $6 \leq \delta \leq 11$.

mentation of the reciprocal (using this approach) required a polynomial approximant of degree at least 10. Since this polynomial is evaluated at runtime, we keep the polynomial with minimal degree $\delta = 10$. As for square root, we could also determine this minimal degree using `guessdegree` function of Sollya, as described in Section 3.2.3 and Listing 3.1.

Once the polynomial degree is determined, the machine-representable polynomial approximant is computed using Remez' algorithm and by truncating each coefficient a_i^* in the $Q_{2.30}$ format. To gain one bit of accuracy on each polynomial coefficient, the coefficients a_i of the polynomial approximant $a(t)$ are stored in k-bit unsigned integers in absolute value such as

$$A_i = |a_i| \cdot 2^{30}.$$

Table 3.16 shows for each truncated coefficient a_i, the sign, the value, and the k-bit unsigned integers A_i encoding $|a_i|$. Remark that, as for the square root, here, we have constructed the

Coefficient	Sign	Value	Encoding integer	Format
a_0	+	2.000000022351741790771484375	`0x80000018`	$Q_{2.30}$
a_1	-	2	`0x80000000`	$Q_{2.30}$
a_2	+	1.999930868856608867645263671875	`0x7ffede0b`	$Q_{2.30}$
a_3	-	1.998307570815086364746093750	`0x7fe44570`	$Q_{2.30}$
a_4	+	1.982783096842467784881591796875	`0x7ee5eb13`	$Q_{2.30}$
a_5	-	1.903202910907566547393798828125	`0x79ce1395`	$Q_{2.30}$
a_6	+	1.659534716978669166564941406250	`0x6a35d11a`	$Q_{2.30}$
a_7	-	1.192411945201456546783447265625	`0x4c507a31`	$Q_{2.30}$
a_8	+	0.626697026193141937255859375	`0x281bcdd8`	$Q_{2.30}$
a_9	-	0.206273326650261878967285156250	`0x0d33950a`	$Q_{2.30}$
a_{10}	+	0.03125	`0x02000000`	Q_{30}

Table 3.16: Coefficients of polynomial approximant for reciprocal implementation.

polynomial $a(t)$ by forcing some coefficients to be powers of two.

Once this machine-representable polynomial is obtained, we compute the certified approximation error bound θ using the supremum norm algorithm of Sollya (see [CL07] or[CJL09]). In this case, the polynomial approximant $a(t)$ approximates the function $2/(1+t)$ over $\mathcal{T}$ with an approximation error bounded by θ defined as follows

$$\alpha(a) \leq \theta \quad \text{and} \quad \theta = 505863026948314408568744590055053329 \times 2^{-144} \approx 2^{-25.39}, \tag{3.61}$$

which is strictly less than the required bound 2^{-25}, and then the conditions in (3.31) are satisfied.

It remains now to evaluate the polynomial P defined as follows:

$$P(t) = 2^{-p-1} + a(t).$$

Instead of evaluating $a(t)$ and then adding 2^{-p-1}, we add first 2^{-p-1} to a_0. Hence for $(k, p, e_{\max}) = (32, 24, 127)$, the degree-0 coefficient of the polynomial $a(t)$ defined in Table 3.16 is defined as follows

$$a_0 = 2.000000052154064178466796875 \quad \text{and} \quad A_0 = \texttt{0x80000038}.$$

Now, given the certified approximation error bound θ in (3.61), it follows that the polynomial $P(t) = a(t)$ has to be evaluated with an evaluation error $\rho(\mathcal{P}) < \eta$, with

$$\begin{aligned} \eta &= \texttt{RoundDownward}(2^{-25} - \theta) \\ &= 158750970944143527883158940085118959 \times 2^{-144} \approx 2^{-27.06}, \end{aligned} \tag{3.62}$$

as defined in (3.37).

Efficient polynomial evaluation

As for square root, the polynomial is evaluated by a straight-line, division free evaluation program (in fixed precision) automatically generated using CGPE (see Chapter 6 for details). In Listing 3.5 below, we present the evaluation program we have kept for implementing the reciprocal in FLIP. This evaluation program has an evaluation latency of 13 cycles. For comparison, using Horner's rule, the evaluation would have been in 40 cycles. Hence, with the evaluation program presented here, we observe a speed-up of 67.5 % (27 cycles). A feasible schedule on ST231 is shown in Table 3.17 below. We observe also on this schedule that three of the four available issues are enough. Notice that the multiplications by powers of two, coefficients A_1 and A_{10}, are simply implemented using shift operations in lines 3 and 21, respectively.

This evaluation program has been validated using Gappa. In this case, we have checked that the evaluation error $\rho(\mathcal{P})$ is strictly less than the certified bound $\eta \approx 2^{-27.06}$ in 3.62. In fact the strict inequality is checked with Gappa through an inequality

$$\rho(\mathcal{P}) \leq \eta - \epsilon,$$

for a small enough, positive $\epsilon = 2^{-300}$.

3.6.5 Implementation of the rounding condition

Until now, we have seen how to compute the approximation v of the exact value ℓ. It remains now to see how to get the correctly-rounded result from this approximation v.

	Issue 1	Issue 2	Issue 3	Issue 4
Cycle 0	r_2	r_{10}		
Cycle 1	r_0	r_{16}		
Cycle 2	r_3	r_8		
Cycle 3	r_7	r_{11}	r_{18}	
Cycle 4	r_{12}	r_{17}		
Cycle 5	r_4	r_{19}		
Cycle 6	r_9	r_{20}		
Cycle 7	r_1	r_5	r_{13}	
Cycle 8	r_{14}			
Cycle 9	r_{21}			
Cycle 10	r_6			
Cycle 11	r_{15}			
Cycle 12	r_{22}			

Table 3.17: Feasible scheduling on ST231, for reciprocal implementation.

```
1  uint32_t __reciprocal__(uint32_t T)
2  {
3    uint32_t r0  = T >> 1;                // 2.30
4    uint32_t r1  = 0x80000038 - r0;       // 2.30
5    uint32_t r2  = mul(T, T);             // 0.32
6    uint32_t r3  = mul(T, 0x7fe44570);    // 2.30
7    uint32_t r4  = 0x7ffede0b - r3;       // 2.30
8    uint32_t r5  = mul(r2, r4);           // 2.30
9    uint32_t r6  = r1 + r5;               // 2.30
10   uint32_t r7  = mul(r2, r2);           // 0.32
11   uint32_t r8  = mul(T, 0x79ce1395);    // 2.30
12   uint32_t r9  = 0x7ee5eb13 - r8;       // 2.30
13   uint32_t r10 = mul(T, 0x4c507a31);    // 2.30
14   uint32_t r11 = 0x6a35d11a - r10;      // 2.30
15   uint32_t r12 = mul(r2, r11);          // 2.30
16   uint32_t r13 = r9 + r12;              // 2.30
17   uint32_t r14 = mul(r7, r13);          // 2.30
18   uint32_t r15 = r6 + r14;              // 2.30
19   uint32_t r16 = mul(T, 0x0d33950a);    // 2.30
20   uint32_t r17 = 0x281bcdd8 - r16;      // 2.30
21   uint32_t r18 = r2 >> 7;               // 2.30
22   uint32_t r19 = r17 + r18;             // 2.30
23   uint32_t r20 = mul(r7, r19);          // 2.30
24   uint32_t r21 = mul(r7, r20);          // 2.30
25   uint32_t r22 = r15 + r21;             // 2.30
26   return r22;
27 }
```

Listing 3.5: Evaluation program for *binary32* reciprocal implementation.

Preliminary definition

We have seen in Section 3.1.2 (for $n < 0$) that the exact result of reciprocal may denormalize (see Property 3.1), that is, the correctly-rounded significand to be computed is

$$m_r = \circ(\ell \cdot 2^{-\lambda_r}).$$

Given that the value $v = \mathcal{P}(t) = (v_{-1}v_0.v_1v_2\cdots v_{k-2})_2$ has a finite binary expansion with $k-2$ fraction bits and that approximates the exact value ℓ from above as in (3.21) such that $-2^{-p} < \ell - v \leq 0$, the correctly-rounded significand satisfies the following condition: $-2^{-p-\lambda_r} < \ell \cdot 2^{-\lambda_r} - v \cdot 2^{-\lambda_r} \leq 0$. Since $\lambda_r \geq 0$, we have $-2^{-p} \leq -2^{-p-\lambda_r}$ and thus

$$-2^{-p} < \ell \cdot 2^{-\lambda_r} - v \cdot 2^{-\lambda_r} \leq 0.$$

The approach is the same as for square root, and the one presented in Section 2.2.4. Let us consider u be the value $v \cdot 2^{-\lambda_r}$ truncated after p fraction bits: $u = (u_{-1}u_0.u_1u_2\cdots u_p)_2$ as in (2.15), and

$$-2^{-p} < \ell \cdot 2^{-\lambda_r} - u < 2^{-p}.$$

The idea is now to compute $\circ(\ell \cdot 2^{-\lambda_r})$ from the approximation u. Unlike for square root, the reciprocal function must be studied for all the four rounding-direction attributes. From Property 3.6, we know that ℓ cannot be halfway between two floating-point numbers. More particularly, in Section 3.1.3, we have seen that the exact result $r = 1/x$ cannot be exactly halfway between two floating-point numbers. Hence, even for trivial input, the value $\ell \cdot 2^{-\lambda_r}$ can never be the middle of two floating-point numbers.

In the rounding algorithm presented in Section 2.2.4 for RoundTiesToEven, to handle the case where $\ell \cdot 2^{-\lambda_r}$ is exactly halfway between two floating-point numbers, we have introduced the condition

$$(u = \ell) \ \wedge \ (u_{p-1} = 0)\,.$$

In this case, $\ell \cdot 2^{-\lambda_r}$ cannot be exactly halfway two floating-point numbers, and the rounding test for RoundTiesToEven can be simplified, as follows, with $m_r = \circ(\ell \cdot 2^{-\lambda_r})$:

$$\begin{aligned} &\text{if } u \geq \ell \cdot 2^{-\lambda_r} \text{ then} \\ &\quad m_r = \texttt{truncate}(u)_{p-1}, \\ &\text{else} \\ &\quad m_r = \texttt{truncate}(u + 2^{-p})_{p-1}. \end{aligned} \tag{3.63}$$

It follows that the implementation of the rounding procedure relies on the implementation of the following conditions:

$$u \geq \ell \cdot 2^{-\lambda_r} \quad \text{or} \quad u > \ell \cdot 2^{-\lambda_r}.$$

The exact value ℓ is not known exactly, and the conditions above cannot be implemented directly. However it turns out that these conditions may be implemented *exactly* using $u \cdot m'_x$ and $2^{1-\lambda_r}$ instead of u and $\ell \cdot 2^{-\lambda_r}$, since $\ell = 2 \cdot (m'_x)^{-1}$, and $u, m'_x > 0$.

To do so, as for square root, recall that V is the k-bit unsigned integer V encoding the computed value v such as $V = v \cdot 2^{k-2}$, and U be the k-bit unsigned integer encoding u such as $U = u \cdot 2^{k-2}$. Since $k > p+2$, by definition of u and v, we deduce that the bit string of U is as follows

$$U = \underbrace{00\cdots00}_{\lambda_r \text{ zeros}} 01 v_1 \cdots v_{p-\lambda_r} \underbrace{00\cdots00}_{k-p-2 \text{ zeros}}\,. \tag{3.64}$$

More particularly by definition of u and v, we deduce that $U = \lfloor (V \cdot 2^{-\lambda_r})/2^{k-2} \rfloor \cdot 2^{k-2}$. Using integer arithmetic, the value U can be computed from V and L_r encoding λ_r by shift right V by L_r bits, and removing the $k-p-2$ last bits of the resulting integer.

```
U = (V >> Lr) & 0xFFFFFFC0;
```

(The computation of L_r is given in Section 3.6.2.) Recall that M_{px} and T are the k-bit unsigned integers encoding m'_x and t_x respectively, as explained in 2.1:

$$M_{px} = m'_x \cdot 2^{k-1} \quad \text{and} \quad T_x = t_x \cdot 2^k.$$

For implementing rounding conditions, let us first define

$$Q = 2^{k-2-\lambda_r}.$$

And before presenting the implementation of these conditions, recall that by definition of `mul`, we have:

$$u \cdot m'_x \cdot 2^{k-3} - 1 < P \leq u \cdot m'_x \cdot 2^{k-3}, \tag{3.65}$$

with $P = \texttt{mul}(U, M_{px})$.

Implementation of the condition $u \geq \ell \cdot 2^{-\lambda_r}$ for RoundTiesToEven

In RoundTiesToEven, the condition does not depend on the sign of the result. Let us see now how to implement this condition introduced in Section 2.2.4.

Property 3.11. *The rounding test $u \geq \ell \cdot 2^{-\lambda_r}$ is true if and only if $P \geq Q$.*

Proof. By definition, since $m'_x > 0$, the condition $u \geq \ell \cdot 2^{-\lambda_r}$ is true, if and only if $u \cdot m'_x \geq 2^{1-\lambda_r}$ is true. We know that $2^{k-3} > 0$, then $u \cdot m'_x \geq 2^{1-\lambda_r}$ is equivalent to $u \cdot m'_x \cdot 2^{k-3}$. From the left inequality of (3.65), we deduce that if $u \cdot m'_x \geq 2^{1-\lambda_r}$ then $P + 1 > Q$ is true, and $P \geq Q$ since P and Q are integers.

On the other hand, if $P \geq Q$, then $P \cdot 2^{3-k} \geq Q \cdot 2^{3-k}$. And using the right inequality of (3.65), we conclude that $u \cdot m'_x \geq 2^{1-\lambda_r}$ and $u \geq \ell \cdot 2^{-\lambda_r}$. □

From Property 3.11, the condition $u \geq \ell \cdot 2^{-\lambda_r}$ is true if and only if $P \geq Q$, and that is if and only if the C condition `P >= Q` holds. For $(k, p, e_{\max}) = (32, 24, 127)$, the rounding condition $u \geq \ell \cdot 2^{-\lambda_r}$ may be implemented using the following piece of C code.

```
Q = 0x40000000 >> Lr;

cond = (P >= Q);
```

Implementation of the rounding conditions for RoundTowardPositive

As shown in Section 2.2.4, for RoundTowardPositive, the rounding condition to be implemented is $u \geq \ell \cdot 2^{-\lambda_r}$ if the result is positive ($s_r = 0$), and $u > \ell \cdot 2^{-\lambda_r}$ if the result is negative ($s_r = 1$). In the same way as for square root, the idea is here to implement exactly the rounding test. To do so, let us define the $Q' \in \{0, 1\}$, as

$$Q' = \begin{cases} 1 & \text{if and only if } (u \cdot m'_x \cdot 2^{k-3} = P \text{ and } s_r = 1), \\ 0 & \text{otherwise.} \end{cases}$$

Hence $Q' = 1$ if and only if $U \cdot M_x \cdot 2^{-k} = P$ and $s_r = 1$ (negative result).

Property 3.12. *Assume that the result is positive ($s_r = 0$). The rounding test $u \geq \ell \cdot 2^{-\lambda_r}$ is true if and only if $P \geq Q + Q'$.*

Proof. Since the result is positive, by definition we know that $Q' = 0$. Hence from Property 3.11, we know that the condition $u \geq \ell \cdot 2^{-\lambda_r}$ is true if and only if $P \geq Q = Q + Q'$, that ends the proof. □

Property 3.13. *Assume that the result is negative ($s_r = 1$). The rounding test $u > \ell \cdot 2^{-\lambda_r}$ is true if and only if $P \geq Q + Q'$.*

Proof. By definition, since $m'_x > 0$, the condition $u > \ell \cdot 2^{-\lambda_r}$ is true, if and only if $u \cdot m'_x > 2^{1-\lambda_r}$ is true. We know that $2^{k-3} > 0$, then $u \cdot m'_x > 2^{1-\lambda_r}$ is equivalent to $u \cdot m'_x \cdot 2^{k-3} > 2^{k-2-\lambda_r}$. Assume first that $Q' = 1$. From (3.65), and the definition of Q and Q', we deduce that the condition $u \cdot m'_x > 2^{1-\lambda_r}$ is equivalent to $P > Q$ that is $P \geq Q + 1 = Q + Q'$.

Assume now that $Q' = 0$. Here we know that $P \neq u \cdot m'_x \cdot 2^{k-3}$ (since $s_r = 1$), thus from (3.65) $P < u \cdot m'_x \cdot 2^{k-3}$, and $P \geq Q$ implies $u \cdot m'_x \cdot 2^{k-3} > Q \cdot 2^{3-k} = 2^{1-\lambda_r}$. On the other hand, if $u \cdot m'_x \cdot 2^{k-3} > Q \cdot 2^{3-k}$ then from (3.65) we conclude that $P + 1 > Q$ that is $P \geq Q = Q + Q'$. □

It remains now to compute the value Q'. In fact, if $U \cdot M_x \cdot 2^{-k} = P$, that means that the computed value u is equal to the exact value $\ell \cdot 2^{-\lambda}$. From Property 3.5 we know that the value ℓ cannot be exactly a floating-point number except for trivial input, and from Property 3.6 ℓ cannot be exactly between two floating-point numbers. Since by definition, $\lambda_r \in \{0, 1, 2\}$, we conclude that $U \cdot M_x \cdot 2^{-k} = P$ if and only if x is a trivial input: $m'_x = 1.0$ and $t_x = 0$. Assuming that the integers S_r and T_x encode the values s_r and t_x, respectively, such as

$$Sr = s_r \cdot 2^{k-1} \quad \text{and} \quad T_x = t_x \cdot 2^k, \tag{3.66}$$

the computation of Q' may be implemented as follows.

```
Qprime = (Tx == 0x0) & (Sr >> 31);
```

As for square root, we may check that $Q + Q'$ fits exactly in k-bit integer since $Q + Q' < 2^k$. Hence from Properties 3.12 and 3.13, the rounding condition is equivalent to $P \geq Q + Q'$, that is, is true if and only if the C condition `P >= Q + Qprime` holds. Finally, the rounding condition can be implemented using the following piece of C code.

```
Q = 0x40000000 >> Lr;

Qprime = (Tx == 0x0) & (Sr >> 31);

cond = (P >= Q + Qprime);
```

Implementation of the rounding conditions for RoundTowardNegative

On the contrary of the RoundTowardPositive rounding-direction attribute, for the RoundTowardNegative, the rounding condition to be implemented is $u \geq \ell \cdot 2^{-\lambda_r}$ if the result is positive ($s_r = 1$), and $u > \ell \cdot 2^{-\lambda_r}$ if the result is negative ($s_r = 0$). To do so, let us now define the $Q' \in \{0, 1\}$ as

$$Q' = \begin{cases} 1 & \text{if and only if } (u \cdot m'_x \cdot 2^{k-3} = P \text{ and } s_r = 0), \\ 0 & \text{otherwise.} \end{cases}$$

Thus, $Q' = 1$ if and only if $U \cdot M_x \cdot 2^{-k} = P$ and $s_r = 0$ (positive result). From Properties 3.12 and 3.13, we deduce the following corollary.

Corollary 3.2. *If the result is positive ($s_r = 0$) then the rounding condition $u > \ell \cdot 2^{-\lambda_r}$ is true if and only if $P \geq Q + Q'$. Otherwise the result is negative ($s_r = 1$) and the rounding condition $u \geq \ell \cdot 2^{-\lambda_r}$ then is true if and only if $P \geq Q + Q'$.*

It remains now to compute the value Q'. In fact, as presented above for RoundTowardPositive, $U \cdot M_x \cdot 2^{-k} = P$ if and only if x is a trivial input: $m'_x = 1.0$ and $t = 0$. Recalling that the integers Sr and T encode the values s_r and t as in (3.66), the computation of Q' may be implemented as follows.

```
Qprime = (Tx == 0x0) & ((~Sr) >> 31);
```

In this case, we may also check that $Q + Q'$ fits exactly in a k-bit integer since $Q + Q' < 2^k$. From Corollary 3.2, the rounding condition is equivalent to $P \geq Q + Q'$: thus this condition is true if and only if the C condition `P >= Q + Qprime` holds. Finally, the rounding condition can be implemented using the following piece of C code.

```
Q = 0x40000000 >> Lr;

Qprime = (Tx == 0x0) & ((~Sr) >> 31);

cond = (P >= Q + Qprime);
```

Implementation of the condition $u > \ell \cdot 2^{-\lambda_r}$ for RoundTowardZero

As for RoundTiesToEven, the rounding condition for RoundTowardZero does not depend on the sign of the result, and is $u > \ell \cdot 2^{-\lambda_r}$. To implement it, let us define $Q' \in \{0, 1\}$ as

$$Q' = \begin{cases} 1 & \text{if and only if } u \cdot m'_x \cdot 2^{k-3} = P, \\ 0 & \text{otherwise.} \end{cases}$$

We have $Q' = 1$ if and only if $U \cdot M_x \cdot 2^{-k} = P$, and using the same approach as before for the other rounding-direction attributes, we conclude that $Q' = 1$ if and only if x is a trivial input: $m'_x = 1.0$ and $t_x = 0$. Using (3.66), the computation of Q' can thus be implemented as follows.

```
Qprime = (Tx == 0x0);
```

From Property 3.13, we can deduce the following corollary.

Corollary 3.3. *For RoundTowardZero, the rounding condition $u > \ell \cdot 2^{-\lambda_r}$ is true if and only if $P \geq Q + Q'$.*

We still may check that $Q + Q'$ fits exactly in a k-bit integer since $Q + Q' < 2^k$. And from Corollary 3.3, the rounding condition is equivalent to $P \geq Q + Q'$: finally this condition is true if and only if the C condition `P >= Q + Qprime` holds. A C implementation of this last rounding condition thus is as follows:

```
Q = 0x40000000 >> Lr;

Qprime = (Tx == 0x0);

cond = (P >= Q + Qprime);
```

Chapter 4

Extension to correctly-rounded division

This chapter extends to division the approach based on the evaluation of a particular bivariate polynomial, introduced in Chapter 3 for the implementation of roots and their reciprocals. Here again, detailed algorithms are given and illustrated with examples of C codes for the binary32 *floating-point format. Unlike roots and their reciprocals, the difficulties are the handling of special inputs (since division is bivariate), the validation of the evaluation program, which is less immediate and cannot be done directly on the whole input interval, and the rounding condition implementations. This implementation of division based on polynomial evaluation is actually about* 1.38 *faster (*1.74 *without subnormal support) compared to the one of FLIP 0.3. Notice that its is also more efficient than iterative method using a specific ST231 instruction (*`divs`*) computing a nonrestoring-like iteration in 1 cycle.*

In this chapter, we propose an extension to division of the uniform approach presented in Chapter 3 for roots and their reciprocals. Generally, floating-point division is less frequently used in numerical applications than addition, subtraction, or multiplication. Indeed its design has often been neglected by developers, that has yielded high latency implementations. In 1997, Oberman and Flynn [OF97a] showed that addition was in 2 to 4 cycles and that multiplication was in 2 to 8 cycles, while division was in 6 to 61 cycles. From this statement, we can observe that low latency implementation of division, in particular on ST231, may still be an issue.

As for nth roots, various methods have actually been used to implement correctly-rounded division, in hardware and software. Among the most classical ones, let us quote *iterative methods*: (non)restoring [OF96], [OF97b], SRT, ..., and *multiplicative methods*: Newton-Raphson and Goldschmidt [Rai06], [PB02]. The former produces one or a few bits of the result per iteration, while the latter refines a first approximation of $1/y$, obtained for example by look-up table [Obe99], small degree polynomial evaluation [Rai06], or specific hardware instruction (`frcpa` in [Mar00], [CHT02], for example). In [Rai06] and FLIP 0.3, the lowest measured latency is of 47 cycles, and has been obtained by exposing instruction-level parallelism of the ST231 through a suitable combination of small degree univariate polynomial evaluation and Goldschmidt's method. Among the most classical methods, we can also quote polynomial-based ones. They are usually implemented through the evaluation of a univariate polynomial of minimal degree [AGS99].

The ST231 processor has a specific instruction, called `divs`, that computes a nonrestoring-like division iteration in one cycle. Therefore, we have implemented (non)restoring iterations. We will show that, using nonrestoring iteration on ST231, with this specific `divs` instruction, leads to an implementation without subnormal support in 39 cycles, that is, already 1.2 times faster than the

available division in FLIP 0.3.

Moreover, for division, the single bivariate polynomial-based methodology of Chapter 3, enables to achieve an implementation in 34 cycles, hence a speedup by a factor of about 1.38 compared to the implementation of FLIP 0.3. If we do not support subnormals numbers, which is actually the case in FLIP 0.3, the implementation falls to 27 cycles, with a speedup by a factor of about 1.74 [JKM+09]. However, unlike for the functions presented in Chapter 3, the difficulties for division rely on the handling of special input, since division is bivariate, the automatic validation of the generated evaluation program using sufficient evaluation error bound, which is less immediate, and the implementation of the rounding condition, since the exact result of the division can be exactly halfway between two floating-point numbers. Concerning the validation, the strategy we have implemented is based on the splitting of the input domain into several subintervals, so that the evaluation program can be validated on each subinterval.

This chapter is organized as follows. Section 4.1 presents the general algorithm, and more particularly explains how to reduce the computation of the division of two floating-point numbers to the evaluation of a bivariate polynomial. Then Sections 4.2 and 4.3 present an efficient way of computing the sign and the exponent of the result, as well as how to handle special inputs. This is mainly optimized for the ST231 processor thanks to a clever use of the binary interchange encoding of input floating-point data. After a reminder of a classically used method in Section 4.4 and its specialization on ST231 using the `divs` instruction, some details on the bivariate polynomial and its validation are given in Section 4.5. Finally Section 4.6 details how to implement the rounding for each of the four rounding-direction attributes.

4.1 General properties of division

This section gives the range reduction of the division function $(x, y) \mapsto x/y$ as well as some useful properties for implementing correct rounding.

Let x and y be two nonzero (sub)normal binary floating-point numbers and let $r = x/y$ be the exact result of division. As mentioned in Section 2.2.1, the goal consists in defining the exact result as $r = (-1)^{s_r} \cdot \ell \cdot 2^d$, with $\ell \in \mathbb{R} \cap [1, 2]$ and $d \in \mathbb{Z}$. Recalling that from Section 1.2.1

$$x = (-1)^{s_x} \cdot m'_x \cdot 2^{e'_x} \quad \text{and} \quad y = (-1)^{s_y} \cdot m'_y \cdot 2^{e'_y},$$

with $s_x, s_y \in \{0, 1\}$, $m'_x, m'_y \in [1, 2 - 2^{1-p}]$, and $e'_x, e'_y \in \{e_{\min} - p + 1, \cdots, e_{\max}\}$, the exact result $r = x/y$ can be expressed as

$$r = (-1)^{s_r} \cdot m'_x/m'_y \cdot 2^{e'_x - e'_y}, \tag{4.1}$$

with $s_r = s_x \oplus s_y$ and $m'_x/m'_y \in (1/2, 2)$. Remark that from Section 2.2.4, we have to ensure that ℓ is a real value that belongs to $[1, 2]$. To do so, let us define the value c as

$$c = \begin{cases} 1, & \text{if } m'_x \geq m'_y, \\ 0, & \text{if } m'_x < m'_y. \end{cases} \tag{4.2}$$

Therefore, using (4.2) together with (4.1), it follows that the exact result $r = x/y$ is

$$r = (-1)^{s_r} \cdot \ell \cdot 2^d, \quad \text{with} \quad \ell = 2^{1-c} \cdot m'_x/m'_y \quad \text{and} \quad d = e'_x - e'_y - 1 + c. \tag{4.3}$$

More particularly, it follows from (4.3) together with the definition of x and y that

$$\ell = s \cdot m'_x/m'_y \quad \text{with} \quad s = 2^{1-c}, \quad \text{and} \quad d \in \{2e_{\min} - p - 1, \ldots, 2e_{\max} + p - 2\}. \tag{4.4}$$

We have given an expression for rewriting the exact result $r = x/y$ as $r = (-1)^{s_r} \cdot \ell \cdot 2^d$, and given a range for d. Let us now determine a range for the exact value ℓ.

Property 4.1. *If $m'_x \geq m'_y$ then $\ell \in [1, 2-2^{1-p}]$, else $\ell \in (1, 2-2^{1-p})$.*

Proof. If $m'_x \geq m'_y$ then $c = 1$, and we deduce from $1 \leq m'_x \leq 2-2^{1-p}$ and $0 < 1/m'_x \leq 1/m'_y \leq 1$ that $1 \leq \ell \leq 2-2^{1-p}$. If $m'_x < m'_y$ then $c = 0$ and $m'_x \leq m'_y - 2^{1-p}$: thus $\ell \leq 2 - 2^{2-p}/m'_y$. Hence, using $m'_x \geq 1$ and $1/m'_y > 1/2$, we obtain $1 < \ell < 2-2^{1-p}$ as desired. □

Using Property 4.1 above and the range for d in (4.4), we deduce that the exact r in (4.3) may underflow or overflow, as shown in Example 4.1 below.

Example 4.1. *Let $x = 2^{e_{min}-p+1}$ and $y = 2^{e_{max}}$. Since $e_{max} = 1 - e_{min}$ and $e_{max} \geq 2$, we have*

$$x/y = 2^{2e_{min}-p+1} < 2^{e_{min}} \quad \text{and} \quad y/x = 2^{2e_{max}-2+p} > \Omega.$$

Hence we deduce that the correctly-rounded result $\circ(r)$ may also denormalize whatever the rounding-direction attribute is. Thus from Section 2.2.1, we have to distinguish three cases:

- if $d > e_{\max}$ then the exact result r overflows;
- if $d \in \{e_{\min}, \ldots, e_{\max}\}$ then r lies in the range of normal floating-point numbers;
- else if $d < e_{\min}$ then r lies in the range of subnormal numbers: $r = (-1)^{s_r} \cdot (\ell \cdot 2^{d-e_{\min}}) \cdot 2^{e_{\min}}$.

Here, we do not discuss the case where $d > e_{\max}$, since it is handled as presented in Section 2.3.2. Assume now that $d \leq 2^{e_{\max}}$. As mentioned in Section 2.2.1, let us define

$$\lambda_r = \max(0, e_{\min} - d) \quad \text{and} \quad e_r = \max(e_{\min}, d), \tag{4.5}$$

with, by definition of the exponent d,

$$\lambda_r \in \{0, \ldots, -e_{\min} + p + 1\}. \tag{4.6}$$

We shall return the correctly rounded result $\circ(r)$ defined as

$$\circ(r) = (-1)^{s_r} \cdot m_r \cdot 2^{e_r} \quad \text{with} \quad m_r = \circ(\ell \cdot 2^{-\lambda_r}) \quad \text{and} \quad e_r \in \{e_{\min}, \ldots, e_{\max}\}. \tag{4.7}$$

We will see in Section 4.2 how to implement the computation of the sign s_r and of the exponent e_r, and in Section 4.6 how to implement m_r. But, let us first give a remark on the overflow case.

Overflow when $\ell = 2$ and $d = e_{\max}$ cannot occur

Recall from Section 2.2.1 that we know that an overflow occurs when $d > e_{\max}$, or when $d = e_{\max}$ and $\ell = 2$. However, it follows from Property 4.1 that $\ell \cdot 2^{-\lambda_r} \leq 2 - 2^{1-p}$, and the case where $\ell = 2$ and $d = e_{\max}$ cannot occur, and no *special implementation case* has to be handled.

Hence, for the division operation, an overflow occurs if and only if $d > e_{\max}$, and this case can be handled as presented in Section 2.3.2.

4.2 Result sign and exponent computation

This section presents how to compute the sign and the exponent of the result in (4.3) from the input x and y encoded into k-bit unsigned integers X and Y, respectively, using the binary interchange encoding described in Section 1.2.2. Recall from Property 2.1 that the k-bit unsigned integer R in (2.21) that encodes the result r is

$$R = S_r + D \cdot 2^{p-1} + M_r,$$

with $S_r = s_r \cdot 2^{k-1}$ and $D = E_r - n_r$ and n_r denotes the "is normal bit" of the result r. Let us now see how to compute the integers S_r and D.

4.2.1 Result sign computation

The computation of the sign of the result s_r is trivial. It is encoded by the integer S_r, obtained by taking the XOR of the sign bits of X and Y:

$$S_r = (X \oplus Y) \wedge 2^{k-1}.$$

For $(k, p, e_{\max}) = (32, 24, 127)$, we get the following piece of C code for computing S_r:

```
Sr = (X ^ Y) & 0x80000000;
```

4.2.2 Result exponent computation

Let us now explain how to compute the exponent D defined as $D = E_r - n_r$. Recall that the k-bit unsigned integers E_x and E_y encoding of the biased exponents of x and y, respectively, are such that

$$E_x = e'_x + \lambda_x - e_{\min} + n_x \quad \text{and} \quad E_y = e'_y + \lambda_y - e_{\min} + n_y.$$

where n_x and n_y are the "is normal" bit of x and y, respectively, and

$$e'_x = e_x - \lambda_x \quad \text{and} \quad e'_y = e_y - \lambda_y,$$

and λ_x and λ_y defined as in (2.1), that is:

$$\lambda_x = MX - w \quad \text{and} \quad \lambda_y = MY - w.$$

Assume now that E_r is the k-bit unsigned integer encoding of the biased value of e_r. Then

$$\begin{aligned} D &= E_r - n_r \\ &\quad e_r - e_{\min}, \end{aligned}$$

since $E_r = e_r - e_{\min} + n_r$ (with n_r the "normal bit" of the result). By definition of e_r in (4.5) as $e_r = \max(e_{\min}, d)$, we finally have

$$D = \max(0, d - e_{\min}), \tag{4.8}$$

where, by definition of d in (4.1)

$$\begin{aligned} d - e_{\min} &= e'_x - e'_y - 1 + c - e_{\min} \\ &= (E_x - E_y) - (n_x - n_y) - (MX - MY) - 1 + c - e_{\min}. \end{aligned}$$

For $(k, p, e_{\max}) = (32, 24, 127)$, using the Listing 2.1 for unpacking input, we obtain the following piece of C code, where D in (4.8) is implemented at line 13. The parenthezisation chosen here tends to expose as much instruction-level parallelism as possible.

```
1  absX = X & 0x7FFFFFFF;              absY = Y & 0x7FFFFFFF;
2
3  MX = max(nlz(absX) , 8);            MY = max(nlz(absY) , 8);
4
5  nx = absX >= 0x00800000;            ny = absY >= 0x00800000;
6
7  Ex = absX >> 23;                    Ey = absY >> 23;
8
9  Mpx = (X << 8) | 0x80000000;        Mpy = (Y << 8) | 0x80000000;
10
11 c = Mpx >= Mpy;
12
13 D  = max(0 , ((Ex - Ey) - (nx - ny)) - ((MX - MY) + (c + 125)));
```

4.3 Special input handling

This section presents how to filter out special inputs, and to determine which special output has to be returned. In fact, we show here an efficient method used to detect special input for *bivariate* functions, that can also be used for addition, subtraction, or multiplication (see Section 2.4, for example, and especially Listing 2.2 where it is used for multiplication).

In SoftFloat [Hau], special input handling is done by extracting the biased exponent of inputs, and considering all the cases of special input: biased exponent equals to 0 or $2^w - 1$. And then the output is selected according to that exponent as well as the sign and the trailing significand fields of the inputs. In [Rai06], the special input handling for bivariate functions like division is also done by extracting the biased exponent. But a first improvement is proposed by detecting if any input is special by using just one `max` instruction and one test. Here, we propose an efficient approach based on the exploitation of the binary interchange format encoding, and more particularly the order of floating-point datum encoding.

4.3.1 IEEE 754 specification of division

Let x and y be two floating-point data as defined in Section 1.2.1. The division operation x/y is defined in Table 4.1. If x or y is a special input, that is, x or y belongs to $\{\pm 0, \pm\infty, \text{NaN},\}$, the IEEE

x/y		y			
		± 0	(sub)normal	$\pm\infty$	NaN
x	± 0	qNaN	± 0	± 0	qNaN
	(sub)normal	$\pm\infty$	$\circ(x/y)$	± 0	qNaN
	$\pm\infty$	$\pm\infty$	$\pm\infty$	qNaN	qNaN
	NaN	qNaN	qNaN	qNaN	qNaN

Table 4.1: Special values for x/y.

754-2008 standard requires that a special value be returned. Since $x/y = (-1)^{s_r} \cdot |x|/|y|$, we just have to determine the special output for $|x|/|y|$, and to adjoin the correct sign. Let us now see how to detect if x or y is a special input, and then how to decide which result shall be returned.

4.3.2 Filtering out special inputs

Let X and Y be the k-bit unsigned integers encoding x and y (with the binary interchange encoding defined in (1.2.2)), and $absX$ and $absY$ those encodings of $|x|$ and $|y|$, respectively. From Tables 4.1 and 1.4, we can observe that $(|x|, |y|)$ (and (x, y) as well) is a special input if and only if

$$absX \text{ or } absY \in \{0\} \cup \{2^{k-1} - 2^{p-1}, \ldots, 2^{k-1}\}.$$

In our context, integer additions and subtractions are done modulo 2^k, so that $absX = 0$ if and only if $absX - 1 = 2^k - 1$. Thus, $(|x|, |y|)$ is a special input if and only if

$$absX - 1 \geq 2^{k-1} - 2^{p-1} - 1 \quad \text{or} \quad absY - 1 \geq 2^{k-1} - 2^{p-1} - 1,$$

that is,

$$\max(absX - 1, absY - 1) \geq 2^{k-1} - 2^{p-1} - 1. \tag{4.9}$$

Remark that on the ST231 processor, this approach may be much faster than testing each kind of special input, since a $\max$ instruction is available, which takes only 1 cycle.

4.3.3 Deciding the result to be returned

Assume now that (x, y) is a special input. For each special case, the IEEE 754-2008 standard [IEE08] specifies that a special value has to be returned, as defined in Table 4.1. We see that the IEEE 754-2008 standard specifies the following output:

1. If $|x| = |y|$, or $|x|$ or $|y|$ is NaN then return a qNaN;
2. else if $|x| < |y|$ then return $\pm\infty$;
3. else if $|x| > |y|$ then return ± 0.

From Table 1.4, in the test in item 1, $|x|$ or $|y|$ is NaN if and only if

$$\max(absX, absY) > 2^{k-1} - 2^{p-1}. \tag{4.10}$$

An example of implementation is given in the following code, for $(k, p, e_{\max}) = (32, 24, 127)$. Here, line 3 implements (4.9), while line 5 implements (4.10).

```
1  absXm1 = absX - 1;            absYm1 = absY - 1;
2
3  if( maxu(absXm1 , absYm1 ) > 0x7F7FFFFF ){
4    Max = maxu(absX , absY);
5    if(absX == absY || Max > 0x7F800000)     // item 1
6      return (Sr | 0x7FC00000) | Max;
7    if(absX < absY) return Sr | 0x7F800000;  // item 2
8    return Sr;                               // item 3
9  }
```

In this piece of C code, when the returned result is a qNaN, we may check that its payload[1] is one of the input payloads, as recommended by the IEEE 754-2008 standard [IEE08, §6.2.3]. This enables to propagate as much information of the input as possible on the result.

4.4 Correctly-rounded division by digit recurrence

Until now we have seen the method used to filter out special inputs, and to compute the pair (exponent, sign) of the result. It remains now to see how to compute the correctly-rounded significand. In this section, we give some key elements for computing this significand using two classical *digit recurrence algorithms*, the *restoring* method and the *nonrestoring* method. Both methods consist in computing one bit of the result per iteration: but the former updates at step j the jth bit if this one is wrong, while the latter updates it at step $j+1$ by incorporating this *restoration* into the computation of the $(j+1)$st bit. The interest of the nonrestoring method relies on the fact that there is a specific instruction on ST231 that (almost) computes a nonrestoring iteration in 1 cycle [ST208b, p. 195].

4.4.1 General principle

The (non)restoring algorithm [EL94] is an iterative process that produces one bit of the result per iteration, such that, after n iterations and a possible final correction step in the case of nonrestoring method

$$0 \leq \ell - \ell[n] < 2^{-n}, \tag{4.11}$$

[1] Recall that we call *payload* the "diagnostic information contained in a NaN, encoded in part of its trailing significand field" [IEE08, p. 4].

where $\ell[n] = \sum_{i=0}^{n} \ell_i \cdot 2^{-i}$ denotes an approximation of the quotient $\ell = \ell_{-1}\ell_0.\ell_1\ell_2\ell_3\ldots$ defined in (4.4) after n iterations. Remark from Property 4.1 that $\ell_{-1} = 0$ and $\ell_0 = 1$.

At each step j, the goal is to select the bit ℓ_j so that the error $\epsilon[j] = \ell - \ell[j]$ remains strictly less than 2^{-j} in magnitude. The way ℓ_j is selected depends on the method (restoring or nonrestoring). But in both cases, it relies on the notion of *partial remainder* $w[j]$ at step j. Recalling that $\ell = 2^{1-c} \cdot m'_x/m'_y$, the value $w[j]$ is defined as

$$w[j] = 2^j \cdot (2^{1-c} m'_x - m'_y \cdot \ell[j]). \tag{4.12}$$

Since $\ell[j+1] = \ell[j] + \ell_{j+1} \cdot 2^{-j-1}$, we have the following recurrence:

$$w[j+1] = 2w[j] - m'_y \cdot \ell_{j+1}, \quad \text{with} \quad w[0] = 2^{1-c} m'_x - m'_y.$$

Finally, it follows from (4.12) that $w[j] = m'_y \cdot 2^j \cdot (\ell - \ell[j])$. Let us now consider both methods and more particularly that $\ell[j]$ is the result of the jth (non)restoring iteration. Then it follows that after n iterations we have

$$-m'_y \leq w[n] < m'_y \quad \text{and} \quad -2^{-n} \leq \epsilon[n] < 2^{-n},$$

and (4.11) holds after a possible correction step since $\epsilon[n]$ may be negative (and in this case, $w[n] < 0$). In fact, $\ell[n]$ represents the first $n+1$ bits of the exact result ℓ. Consequently, to compute the correctly-rounded value $\circ(\ell \cdot 2^{-\lambda_r})$, we observe that p iterations are enough.

Note that we will see in Property 4.3, in Section 4.6.2 below, that the exact value ℓ cannot be exactly halfway between two consecutive floating-point numbers. Hence, at step p, it follows from Property 4.3 (before the possible correction step in the case of nonrestoring method) that the error $\epsilon[p]$ cannot equal 2^{-p} in magnitude, and thus

$$|w[p]| < m'_y \quad \text{and} \quad |\epsilon[p]| < 2^{-p}. \tag{4.13}$$

4.4.2 Restoring division

Iteration description

The restoring method uses the quotient-digit set $\{0, 1\}$ to produce at each step j the bit ℓ_j. At step $j+1$, the principle consists in considering that $\ell_{j+1} = 1$ and computing a *tentative* partial remainder

$$\tilde{w}[j+1] = 2w[j] - m'_y.$$

All the quotient bits are nonnegative. Thus at each step j, the partial remainder $w[j]$ must also be nonnegative: we have to ensure that

$$0 \leq w[j+1] < m'_y.$$

Here, if $\tilde{w}[j+1] \geq 0$ then we conclude that $w[j+1] = \tilde{w}[j+1]$ and $\ell_j = 1$. Otherwise, if $\tilde{w}[j+1] < 0$, then ℓ_j should have been set to 0 instead of 1: we *restore* the quotient-digit ℓ_j to 0 and compute the new partial remainder: $w[j+1] = 2w[j] = \tilde{w}[j+1] + m'_y$. Indeed, at each step j, the quotient-digit is selected according to the sign of the partial remainder $w[j]$. The restoring division algorithm is presented in Algorithm 4.1.

Restoring iteration implementation

Let N and M be the k-bit unsigned integers encoding of $2^{1-c} \cdot m'_x$ and m'_y such that

$$N = 2^{p-c} \cdot m'_x \quad \text{and} \quad M = 2^{p-1} \cdot m'_y.$$

```
  // Initialization
1 w[0] ← 2^{1−c} m'_x − m'_y ;
2 ℓ_0 ← 1 ;

  // Restoring iteration
3 for j ← 0 to p − 1 do
4     w̃[j + 1] ← 2w[j] − m'_y ;        // tentative partial remainder computation
5     if w̃[j + 1] ≥ 0 then
6         w[j + 1] ← w̃[j + 1];
7         ℓ_{j+1} ← 1;
8     else
9         w[j + 1] ← w̃[j + 1] + m'_y ;                          // restoration
10        ℓ_{j+1} ← 0;
11    end
12    ℓ[j + 1] ← ℓ[j] + ℓ_{j+1} · 2^{−j−1};
13 end
```

Algorithm 4.1: Restoring division for computing $\ell_0.\ell_1\ell_2\ldots\ell_{p-1}\ell_p$.

By definition, we know that $M_{px} = m'_x \cdot 2^{k-1}$, and since m'_x has at most $p-1$ fraction bits, M_{px} is a multiple of 2^{k-p}. (The same relationship holds between m'_y and M_{py}.) Hence we obtain

$$N = M_{px} \cdot 2^{p-k+1-c} \quad \text{and} \quad M = M_{py} \cdot 2^{p-k},$$

both of which can be computed exactly.

For $(k, p, e_{\max}) = (32, 24, 127)$ and given the integers M_{px}, M_{py}, and $W_j = w[j] \cdot 2^{p-1}$, the restoring iteration may be implemented using the piece of C code in Listing 4.1. Here at step j,

```
1  uint32_t N, M, L, Wj;
2  int32_t  tj;
3
4  // Initialization
5  N = Mpx >> (7 + c);          M = Mpy >> 8;
6
7  Wj = N - M;                   // j = 0
8  L  = 1;
9
10 // ...
11
12 // Restoring iteration j
13 tj = (Wj << 1) - M;           // tentative partial remainder computation
14
15 if(tj >= 0){
16   L = (L << 1) | 1;
17   Wj = tj;
18 }else{
19   L = L << 1;                 // restoration
20   Wj = tj + M;
21 }
```

Listing 4.1: Restoring division iteration.

the unsigned integer L encodes $\ell[j]$ so that $L = \ell[j] \cdot 2^j$. This iteration may be simplified, using

only logical operators, and lines 12 to 21 of Listing 4.1 can be replaced by those of Listing 4.2 below. Remark that this second iteration can be useful on architectures that do not have any

```
// Restoring iteration j
tj = (Wj << 1) - M;          // tentative partial remainder computation

L = (L << 1) | (tj >= 0);    // restoration, if tj < 0

Wj = tj + ((0x0 - (tj < 0)) & M);
```

Listing 4.2: Restoring division iteration using logical operations.

efficient mechanism of conditional branches reduction. However, on ST231, it leads to an increase of latency, especially for the computation of `Wj`.

4.4.3 Nonrestoring division

From restoring to nonrestoring iterations

In the restoring division, we can observe that restoration at step j may be fused with the tentative partial remainder computation at step $j+1$. Indeed, consider $w[0] = 2^{1-c} \cdot m'_x - m'_y$ and let us start by computing $w[1] = 2w[0] - m'_y$ by assuming $\ell_1 = 1$. Then, at each iteration j, we have the following:

- If $w[j] \geq 0$, the tentative $\ell_j = 1$ at the previous step was correct, and we just compute the partial remainder for step $j+1$ by assuming $\ell_{j+1} = 1$:

$$w[j+1] = 2w[j] - m'_y \quad \text{(tentative partial remainder computation).}$$

- If $w[j] < 0$, that means that ℓ_j should not have been set to 1: we fuse the restoration of ℓ_j to 0, and compute the new value $w[j]$, with the computation of the tentative partial remainder for step $j+1$ by assuming $\ell_{j+1} = 1$:

$$w[j+1] = 2w[j] - m'_y \quad \text{(tentative partial remainder computation)}$$

with

$$w[j] = w[j] + m'_y \quad \text{(restoration),}$$

that is,

$$w[j+1] = 2w[j] + m'_y.$$

Compared to restoring division, nonrestoring division produces ℓ_p at step $p+1$, since at step p we just assume that $\ell_p = 1$. Thus a final correction step is needed if $w[n] < 0$. The nonrestoring division is presented in Algorithm 4.2.

Nonrestoring iteration implementation

Using the same definitions as those used for the implementation of restoring iteration, for $(k, p, e_{\max}) = (32, 24, 127)$, the nonrestoring iteration may be implemented using the piece of C code in Listing 4.3 below.

```
   // Initialization
1  w[0] ← 2^{1−c} m'_x − m'_y ; ℓ_0 ← 1 ;
2  w[1] = 2w[0] − m'_y ;  // tentative partial remainder computation at step 1

   // Nonrestoring iteration
3  for j ← 1 to p − 1 do
4      if w[j] ≥ 0 then
           /* tentative partial remainder computation at step j + 1      */
5          w[j + 1] ← 2w[j] − m'_y;
6          ℓ_j ← 1;
7      else
8          w[j + 1] ← 2w[j] + m'_y ;
9          ℓ_j ← 0 ;
10     end
11     ℓ[j] ← ℓ[j − 1] + ℓ_j · 2^{−j};
12 end

   // Correction step
13 if w[p] ≥ 0 then
14     ℓ_p ← 1;
15 else
16     w[p] ← 2w[p] + m'_y ;                          // final correction step
17     ℓ_p ← 0;
18 end
19 ℓ[p] ← ℓ[p − 1] + ℓ_p · 2^{−p};
```

Algorithm 4.2: Nonrestoring division for computing $\ell_0.\ell_1\ell_2\ldots\ell_{p-1}\ell_p$.

```
uint32_t N, M, L, Wj;

// Initialization
N = Mpx >> (7 + c);          M = Mpy >> 8;

Wj = N - M;                   // j = 0
L  = 1;

Wj = (Wj << 1) - M;           // tentative partial remainder at step 1

// ...

// Nonrestoring iteration j
if(Wj >= 0){
  Wj = (Wj << 1) - M;         // tentative partial remainder at step j+1
  L  = (L << 1) | 1;
}else{
  Wj = (Wj << 1) + M;
  L = L << 1;
}

// ...

// Correction step
if(Wj >= 0){
  L = (L << 1) | 1;
}else{
  Wj = (Wj << 1) + M;         // final correction step
  L = L << 1;
}
```

Listing 4.3: Nonrestoring division iteration.

4.4.4 Nonrestoring division on the ST231 processor

On the ST231 processor core, a specific instruction `divs` is available [ST208b, p. 195]. This instruction has a latency of 1 cycle, and almost implements the above nonrestoring iteration. By "almost", we mean that, given $w[j]$ and m'_y, `divs` returns $w[j+1]$ and $q_j = \neg\ell_j$ instead of $w[j+1]$ and ℓ_j, where $\neg\ell_j$ denotes the negation of the bit ℓ_j. Combined with `addcg` (see [ST208b, p. 139]) that adds in 1 cycle an integer with a carry (bit $\ell_j = \neg q_j$), the lines 3 to 12 of Algorithm 4.2 can be reduced to those of Algorithm 4.3 below. The nonrestoring iteration of Listing 4.3 above can be

```
  // Nonrestoring iteration based on divs instruction
1 for j ← 1 to p − 1 do
2     (w[j + 1], q_j) ← divs(w[j], m'_y) ;
3     ℓ[j] ← addcg(ℓ[j − 1], ¬q_j) ;
4 end
```

Algorithm 4.3: Nonrestoring iteration using the `divs` and `addcg` instructions.

replaced by those of Listing 4.4 below.

```
1  uint32_t N, M, L, Wj, Q;
2
3  // Initialization
4  N = Mpx >> (7 + c);          M = Mpy >> 8;
5
6  Wj = N - M;                   // j = 0
7  Q  = 0xFFFFFFFE;
8
9  Wj = (Wj << 1) - M;           // tentative partial remainder at step 1
10
11 // ...
12
13 // Nonrestoring iteration j
14 __DIVS(Wj,qj,Wj,M,0); __ADDCG(Q,qj,Q,Q,qj);
15
16 // ...
17
18 // Correction step
19 L = ~Q;
20
21 if(Wj >= 0){
22   L = (L << 1) | 1;
23 }else{
24   Wj = (Wj << 1) + M;         // final correction step
25   L = L << 1;
26 }
```

Listing 4.4: Nonrestoring division iteration using the `divs` and `addcg` instructions.

In Listing 4.4, the instruction `divs` computes $q_j = \neg\ell_j$ instead of ℓ_j (see line 14). Therefore, instead of computing $\ell_j = \neg q_j$ at each step, the implementation is done using an integer Q defined

as follows at step j

$$\begin{aligned} Q &= \sim (\ell[j] \cdot 2^j) \\ &= 2 \cdot \underbrace{\big(\sim (\ell[j-1] \cdot 2^{j-1}) \big)}_{Q \text{ at step } j-1} + \neg \ell_j. \end{aligned}$$

The bit string of Q corresponds to the negation of the one of L. And before the final correction step, we need to reverse all the bits of Q to get L (see line 19).

As we can see in Figure 4.1 below, the execution of the `divs` instruction at step j can be launched in the same time as the `addcg` instruction of step $j-1$. Thus, assuming at least 2 parallel issues, the full execution of the nonrestoring algorithm is $p-1$ iterations of 1 cycle each, plus 1 cycle for the last `addcg` instruction, that is, p cycles.

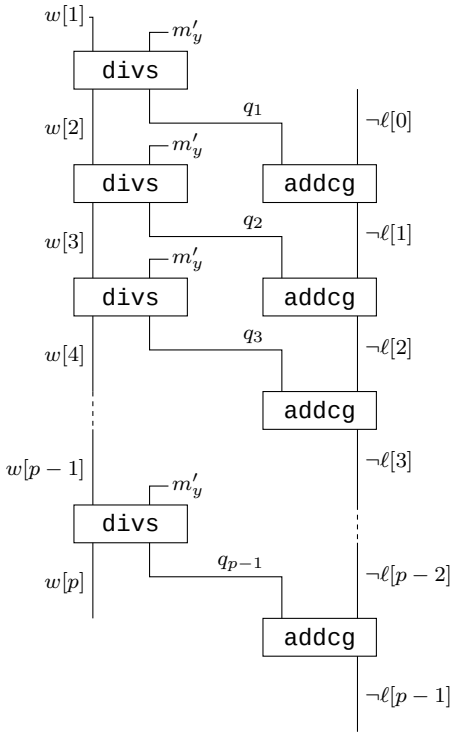


Figure 4.1: Flowchart for nonrestoring iteration using `divs` and `addcg` instructions.

Listing 4.5 below displays the ST200 assembly code corresponding the C code in Listing 4.4. On this assembly code, we observe that once we have computed W_j (cycle 10), one nonrestoring iteration is computed every cycle.

4.4.5 How to achieve correct rounding?

Until now, we have seen how to compute an approximation $\ell[p]$ of ℓ with p fraction bits. Let us now see how to achieve correct rounding.

With subnormal numbers support

Once we have computed $\ell[p]$, which corresponds to exact result ℓ truncated after p fraction bits, it remains to compute the correctly-rounded value of $\ell \cdot 2^{-\lambda_r}$.

```
cmpgeu $r31 = $r29, $r27              ## (cycle 5)
shru $r34 = $r27, 8                   ## (cycle 5)  ## M = Mpy >> 8
and $r38 = $r38, 255                  ## (cycle 5)
;; ## (bundle 5)

add $r36 = $r31, 7                    ## (cycle 6)  ## 7+c
sub $r30 = $r30, $r31                 ## (cycle 6)
;; ## (bundle 6)

shru $r36 = $r29, $r36                ## (cycle 7)  ## N = Mpx >> (7+c)
shru $r29 = $r29, $r31                ## (cycle 7)
;; ## (bundle 7)

sub $r36 = $r36, $r34                 ## (cycle 8)  ## Wj = N - M
shru $r29 = $r29, 1                   ## (cycle 8)
;; ## (bundle 8)

shl $r36 = $r36, 1                    ## (cycle 9)  ## Wj << 1
;; ## (bundle 9)

sub $r36 = $r36, $r34                 ## (cycle 10) ## Wj = (Wj << 1) - M
;; ## (bundle 10)

divs $r33, $b6 = $r36, $r34, $b5      ## (cycle 11) ## __DIVS(Wj,qj,Wj,M,0)
;; ## (bundle 11)

addcg $r28, $b6 = $r35, $r35, $b6     ## (cycle 12) ## __ADDCG(Q,qj,Q,Q,qj)
divs $r32, $b4 = $r33, $r34, $b5      ## (cycle 12) ## __DIVS(Wj,qj,Wj,M,0)
cmpgeu $r35 = $r25, 8388608           ## (cycle 12)

;; ## (bundle 12)
divs $r23, $b3 = $r32, $r34, $b5      ## (cycle 13) ## __DIVS(Wj,qj,Wj,M,0)
addcg $r24, $b4 = $r28, $r28, $b4     ## (cycle 13) ## __ADDCG(Q,qj,Q,Q,qj)
shru $r32 = $r16, 23                  ## (cycle 13)
sub $r35 = $r35, $r37                 ## (cycle 13)
;; ## (bundle 13)

## ...

addcg $r31, $b6 = $r36, $r36, $b6     ## (cycle 33) ## __ADDCG(Q,qj,Q,Q,qj)
divs $r22, $b5 = $r33, $r34, $b5      ## (cycle 33) ## __DIVS(Wj,qj,Wj,M,0)
;; ## (bundle 33)

cmpge $r22 = $r22, $r0                ## (cycle 34)
addcg $r21, $b5 = $r31, $r31, $b5     ## (cycle 34) ## __ADDCG(Q,qj,Q,Q,qj)
;; ## (bundle 34)

xor $r21 = $r21, -1                   ## (cycle 35) ## L = ~Q
;; ## (bundle 35)                                  ##   = Q XOR 0xFFFFFFFF
```

Listing 4.5: Assembly code for nonrestoring division on ST231, using `divs` and `addcg`.

Recall from (4.11) that $0 \leq \ell - \ell[p] < 2^{-p}$. As required in Section 2.2 for correct rounding computation, let us compute $\ell[p] \cdot 2^{-\lambda_r}$ such that

$$0 \leq \ell \cdot 2^{-\lambda_r} - \ell[p] \cdot 2^{-\lambda_r} < 2^{-p-\lambda_r} \leq 2^{-p}.$$

In this case the rounding algorithms are the same (according to rounding-direction attribute) as those presented in Section 2.2.4 where the exact result may have an infinite binary expansion.

We remark here that for nonrestoring division, the correction step ensures that the error $\epsilon[p]$ is non negative. Without correction step, we just know from (4.13) that $|\epsilon[p]| < 2^{-p}$. However, from Section 2.2.4, we observe that this bound is enough to deduce correct rounding. That is, when implementing division by the nonrestoring method with subnormal support, the correction step is not needed and we just have to set $\ell_p = 1$. Consequently, the lines 13 to 19 of Algorithm 4.2 can be replaced by $\ell[p] \leftarrow \ell[p-1] + 2^{-p}$.

Improvement when subnormal numbers are not supported

The computed value $\ell[p]$ represents the first $p+1$ bits of the exact result $\ell \in [1, 2)$:

$$\ell[p] = 1.\ell_1\ell_2 \ldots \ell_p.$$

We will see in Property 4.3, in Section 4.6.2 below, that the exact result cannot be exactly halfway between two floating-point numbers. Consequently, the case (guard bit g, sticky bit s) = (1,0) cannot occur. We can thus reuse the rounding algorithms of Section 2.2.3 and even simplify them to exploit the fact that (g,s) $\neq$ (1,0).

4.4.6 Performances of the (non)restoring algorithms on the ST231 processor

We have implemented the three iterations above. Table 4.2 gives the performances of these algorithms on the ST231 processor.

	Latency (# cycles)	Number of integer instructions	IPC	Code size (bytes)
Restoring	114 [101]	246 [221]	2.16 [2.18]	1004 [916]
Nonrestoring	96 [82]	241 [214]	2.51 [2.61]	1000 [888]
Nonrestoring with `divs`	52 [39]	128 [108]	2.46 [2.77]	548 [464]

Table 4.2: Performances of (non)restoring iterations with [without] subnormal numbers in RoundTiesToEven, on the ST231.

From Table 4.2, we make the following remarks.

- Our implementation using nonrestoring iteration is already about 1.2 times faster than the one using restoring iteration.

- Using the specific `divs` iteration leads to an implementation about twice faster than the one using "naive" nonrestoring iteration, with twice fewer instructions. Therefore, in RoundTiesToEven and without subnormal numbers, this new implementation with `divs` instruction is already faster by a factor of 1.2 than the one of FLIP 0.3 based on Goldschmidt iteration, which was in 47 cycles (see Table 1, Introduction).

4.5 Bivariate polynomial evaluation and validation

We have seen in Section 4.4 how to compute the correctly-rounded significand $m_r = \circ(\ell \cdot 2^{-\lambda_r})$ with two digit-recurrence algorithms. Now let us study how to obtain it by extending the approach based on bivariate polynomial evaluations that we have introduced in Section 3.2 for nth roots. On ST231, this approach will prove to be much faster and makes the `divs` instruction not really useful.

4.5.1 Division via bivariate polynomial evaluation

The goal is to compute a *one-sided approximation* v of $\ell = 2^{1-c} \cdot m'_x/m'_y$ that approximates ℓ from above, such that $-2^{-p} < \ell - v \leq 0$, which is implied by

$$\left|\ell + 2^{-p-1} - v\right| < 2^{-p-1}. \tag{4.14}$$

The value v will be the result of a bivariate polynomial evaluation, where the first variable depends on $2^{1-c} \cdot m'_x$ (to handle range reduction), and the second variable depends on m'_y. Thus, by truncating v, we can then deduce a value u approximating the real value ℓ such that $|\ell \cdot 2^{-\lambda_r} - u| < 2^{-p}$, as required in Section 2.2.4 for ensuring correct rounding.

Bivariate polynomial approximation

To compute the value v, the first step consists in interpreting the exact value $\ell + 2^{-p-1}$ as the exact result $F(s^*, t^*)$, where

$$F(s,t) = 2^{-p-1} + s/(1+t) \quad \text{and} \quad (s^*, t^*) = (2^{1-c} \cdot m'_x, m'_y - 1). \tag{4.15}$$

We may check that

$$s^* \in \mathcal{S} = [1, 2 - 2^{1-p}] \cup [2, 4 - 2^{3-p}] \quad \text{and} \quad t^* \in \mathcal{T} = [0, 1 - 2^{1-p}]. \tag{4.16}$$

The upper bound of the second interval of $\mathcal{S}$ comes from the fact that $c = 0$ implies $m'_x \leq 2 - 2^{2-p}$.

Once we have defined the function F, since it cannot be evaluated directly on a computer, the second step consists in approximating it over $\mathcal{S} \times \mathcal{T}$ by a single bivariate polynomial P. The function F is linear with respect to s, and thus we can reduce to univariate approximation by taking

$$P(s,t) = 2^{-p-1} + s \cdot a(t), \tag{4.17}$$

with $a(t)$ a polynomial approximant of smallest degree of the function $1/(1+t)$ over $\mathcal{T}$. Note that $P(s,t)$ has the same special bivariate form as in Chapter 3. Now, the evaluation of the polynomial P can be done using only additions, subtractions, and multiplications.

Finally this bivariate polynomial P is thus evaluated at (s^*, t^*) by an efficient finite precision evaluation program $\mathcal{P}$, to obtain the value $v = \mathcal{P}(s^*, t^*)$. As in Chapter 3, these steps entail an *approximation error* $\alpha(a)$ and an *evaluation error* $\rho(\mathcal{P})$ given by

$$\alpha(a) = \max_{t \in \mathcal{T}} |1/(1+t) - a(t)| \quad \text{and} \quad \rho(\mathcal{P}) = \max_{(s,t) \in \mathcal{S} \times \mathcal{T}} |P(s,t) - \mathcal{P}(s,t)|. \tag{4.18}$$

Notice that, unlike for nth roots, the point (s^*, t^*) here is representable exactly on a finite (say, k) number of bits. This is why the rounding error $\gamma(s)$ of Chapter 3 does not appear here.

Sufficient condition determination

As we have already seen, the main idea of this approach consists now in determining in an automatic way a polynomial approximant as well as an evaluation program, such that the errors defined in (4.18) ensure that (4.14) holds. The following property gives some sufficient conditions on $\alpha(a)$ and $\rho(\mathcal{P})$ for (4.14) to hold.

Property 4.2. *Given ℓ, v, a and $\alpha(a)$, $\mathcal{P}$ and $\rho(\mathcal{P})$ defined above, if*

$$(4 - 2^{3-p}) \cdot \alpha(a) + \rho(\mathcal{P}) < 2^{-p-1}, \tag{4.19}$$

then (4.14) holds.

Proof. For all (s^*, t^*), we have

$$\begin{aligned} \left|\ell + 2^{-p-1} - v\right| &= |F(s^*,t^*) - \mathcal{P}(s^*,t^*)| \\ &\leq |F(s^*,t^*) - P(s^*,t^*)| + |P(s^*,t^*) - \mathcal{P}(s^*,t^*)| \quad \text{(triangular inequality)} \\ &= |s^*| \cdot |1/(1+t^*) - a(t^*)| + |P(s^*,t^*) - \mathcal{P}(s^*,t^*)|, \quad \text{using (4.15) and (4.17)} \\ &\leq (4 - 2^{3-p}) \cdot \alpha(a) + \rho(\mathcal{P}), \quad \text{using (4.16) and (4.18).} \end{aligned}$$

The conclusion follows immediately from the upper bound in (4.14). □

Since by definition $\rho(\mathcal{P}) \geq 0$, it follows from (4.19) that the approximation error $\alpha(a)$ should satisfy

$$\alpha(a) < 2^{-p-1}/(4 - 2^{3-p}). \tag{4.20}$$

Let θ be a dyadic number such that $\theta < 2^{-p-1}/(4 - 2^{3-p})$. Once we have built a polynomial approximant a such that $\alpha(a) \leq \theta$, it remains to find an evaluation program $\mathcal{P}$, such that

$$\rho(\mathcal{P}) < 2^{-p-1} - (4 - 2^{3-p}) \cdot \theta.$$

Recall that we want to implement mathematical functions with certified C code. Unlike for nth roots in Chapter 3, here, the bound $2^{-p-1} - (4 - 2^{3-p}) \cdot \theta$ is a rational number and can in principle be computed exactly. Hence, it suffices that $\rho(\mathcal{P})$ be such that

$$\rho(\mathcal{P}) < \eta \quad \text{with} \quad \eta = 2^{-p-1} - (4 - 2^{3-p}) \cdot \theta. \tag{4.21}$$

Note that since θ is a dyadic number, η is also a dyadic number.

Automatic generation of polynomial coefficients and certified bounds

We can derive from the script in Listing 3.1, Chapter 3, a Sollya script that computes the polynomial approximant $a(t)$ together with a certified approximation error bound θ, and the certified evaluation error η, so that (4.14) holds. Table 4.3 below shows the degree of the polynomial approximant and the certified error bounds computed with the script in Listing 4.6 below, for the *binary32* floating-point formats.

Degree δ	Approximation error bound θ	Evaluation error bound η
10	$\approx 2^{-27.41}$	$\approx 2^{-26.99}$

Table 4.3: Degree of polynomial approximant $a(t)$, and certified error bounds θ and η, for our *binary32* division implementation.

```
div = proc(k,p){
  // Definition of the function f
  f = 1/(1+x);

  // Definition of the interval T
  T = [0,1-2^(1-p)];

  // Computation of the approximation error bound, defined in (4.20)
  approx = 2^(-p-1)/(4 - 2^(3-p));

  // Determination of the minimal degree δ=delta
  dinterval = guessdegree(f,T,approx);    // Sollya's guessdegree function
  delta = inf(dinterval);

  minimal = 0; while( minimal == 0 ) do {
    // Computation of the Remez's polynomial approximant
    astar = remez(f,delta,T,1,1e-7);        // Sollya's remez function

    // Determination of the size of the integer part of the coefficient
    Qf = k - GetIntegerPartSize(astar,p);

    // Truncation of each coefficient on Qf fraction bits
    a = TruncatePoly(astar,Qf);    // returns the truncated Reme'z polynomial

    // Computation of the certified error bound θ=theta
    diam=1e-8!; thetainterval = infnorm(f-a,T);  // Sollya's infnorm function
    theta = sup(thetainterval);

    // Checking if theta satisfies the required bound θ < 2^(-p-1)/(4 - 2^(3-p))
    if( theta >= approx ) then {
      delta = delta + 1;
    } else {
      minimal = 1;
    };
  };

  // Computation of the certified evaluation error bound η=eta in (4.21)
  eta = 2^(-p-1) - (4 - 2^(3-p)*theta);

  return [|k,p,f,T,approx,delta,astar,a,Qf,theta,eta|];
};
```

Listing 4.6: Sollya script for automatic generation of polynomial coefficients for division implementation.

On this example, the conditions (4.20) and (4.21) are only sufficient conditions, and we may find some cases for which they may be pessimistic. More particularly, given a polynomial approximant a, we may build an evaluation program $\mathcal{P}$ for which the condition (4.21) on evaluation error is not satisfied, whereas the evaluation is accurate enough to ensure correct rounding. We will see further that in this case, we can determine an evaluation error bound by interval subdivision. That means, instead of considering this bound on the whole input domain $\mathcal{T}$, it consists in splitting up this domain into several subintervals $\mathcal{T}^{(i)}$, and checking whether this condition is satisfied on each subinterval. This will be detailed in Section 4.5.3.

4.5.2 Example of implementation for the *binary32* format

We consider here the special case of the implementation of the *binary32* division, that is, for $(k, p, e_{\max}) = (32, 24, 127)$. As for nth roots in Chapter 3, the implementation of the evaluation program to compute the value v approximating ℓ consists of three steps:

1. determination of a polynomial approximant a such that the approximation error bound (4.20) is satisfied and computation of the certified approximation error bound θ;
2. generation of an efficient evaluation program $\mathcal{P}$, that exposes as much instruction-level parallelism as possible and that reduces evaluation latency;
3. validation of this evaluation program to ensure that the evaluation error bound (4.21) holds.

As in Chapter 3, the first step is done using the software environment Sollya [Che09], [Lau08], [CL], while the third is done using Gappa [Mel], [Mel06]. In this chapter, we do not discuss the problem of generating the evaluation program $\mathcal{P}$. We just consider that we have an evaluation program in fixed-point arithmetic as in Listing 4.7 below, generated with CGPE [Rev] presented in Chapter 6. In the following, we present the way used to build the polynomial approximant a, while Section 4.5.3 shows how to validate the generated program.

Polynomial approximant computation

The polynomial P represents the main part of the full division code, and at least on the ST231 its evaluation dominates the cost of the division implementation. The polynomial P defined as

$$P = 2^{-25} + s \cdot a(t)$$

has thus to be of minimal degree, and the polynomial approximant a as well. Here, from (4.20) the polynomial a must approximate the function $1/(1+t)$ over $\mathcal{T} = [0, 1 - 2^{-23}]$ with an absolute approximation error strictly less than $2^{-25}/(4 - 2^{-21}) \approx 2^{-26.99}$.

Recall that the *minimax* polynomial of degree δ with respect to $1/(1+t)$ over $[0, 1 - 2^{-23}]$ is the unique $a^* \in \mathbb{R}[t]_\delta$, where $\mathbb{R}[t]_\delta$ denotes as in Chapter 3 the set of univariate real polynomials of degree at most $\delta \in \mathbb{N}$, such that

$$\alpha(a^*) \leq \alpha(a) \quad \text{and} \quad a \in \mathbb{R}[t]_\delta.$$

Table 4.4 below shows the estimation of $\alpha(a^*)$, when δ ranges from 6 to 11. For each degree δ, the polynomial has been computed using the Remez' algorithm (`remez`) and $\alpha(a^*)$ has been estimated with the certified supremum norm algorithm (`infnorm`) of Sollya. From Table 4.4, we deduce that the minimal required degree is $\delta = 10$, which is the minimal degree for which the required bound (4.20) is satisfied. This lower bound can also be computed using the `guessdegree` function of Sollya.

δ	6	7	8	9	10	11
$\alpha(a^*)$	$2^{-17.25}$	$2^{-19.80}$	$2^{-22.34}$	$2^{-24.88}$	$2^{-27.42}$	$2^{-29.94}$

Table 4.4: Numerical estimation of $\alpha(a^*)$, for $6 \leq \delta \leq 11$.

On the ST231 processor, the evaluation of the polynomial is done by a finite precision program in fixed-point arithmetic. The polynomial coefficients are stored in absolute value in 32-bit unsigned integers in the $Q_{0.32}$ format. Table 4.5 below shows, for each coefficient a_i, the sign, the value, the 32-bit integer $A_i = |a_i| \cdot 2^{32}$ (in hexadecimal representation) that encodes $|a_i|$, and its format. Since the register size is limited to 32 bits, as for square root, storing the coefficients in

Coefficient	Sign	Value	Encoding integer	Format
a_0	+	0.99999999441206455230712890625	`0xfffffe8`	$Q_{0.32}$
a_1	-	0.99999855994246900081634521484375	`0xffffe7d7`	$Q_{0.32}$
a_2	+	0.99993782653473317623138427734375	`0xfffbece7`	$Q_{0.32}$
a_3	-	0.99894478521309792995452880859375	`0xffbad86f`	$Q_{0.32}$
a_4	+	0.99068035138770937919616699218 75	`0xfd9d3a3e`	$Q_{0.32}$
a_5	-	0.95079298713244497776031494140625	`0xf3672b51`	$Q_{0.32}$
a_6	+	0.83134166034869849681854248046875	`0xd4d2ce9b`	$Q_{0.32}$
a_7	-	0.60248205438256263732910156 25	`0x9a3c4390`	$Q_{0.32}$
a_8	+	0.32168737309984862804412841796875	`0x525a1a8b`	$Q_{0.32}$
a_9	-	0.10831562872044742107391357421875	`0x1bba92b3`	$Q_{0.32}$
a_{10}	+	0.01688681519590318202972412109375	`0x0452b1bf`	$Q_{0.32}$

Table 4.5: Coefficients of the polynomial approximant used for *binary32* division.

absolute value leads to a gain of one bit of accuracy on the coefficients. Most of the time, the truncated coefficient polynomial is less accurate than the real one. In our case, the truncated Remez' polynomial a approximates the function $1/(1+t)$ over $\mathcal{T}$ with an approximation error less than θ such that

$$\alpha(a) \leq \theta = 3 \cdot 2^{-29} \approx 2^{-27.41},$$

which is actually less than the bound $2^{-25}/(4 - 2^{-21}) \approx 2^{-26.99}$ in (4.20). Figure 4.2 shows the approximation error of the polynomial a for $t \in \mathcal{T}$.

If the coefficient polynomial approximant had been implemented in signed value in unsigned integers in the signed format $Q_{0.31}$ (we should have used one bit for the sign), the approximation error θ would have been $\theta \approx 2^{-27.39}$, that is, slightly larger than for the polynomial approximant a, and the evaluation should have been slightly more accurate.

Extracting the evaluation point (s^*, t^*)

Before explaing how to evaluate the polynomial P, let us see how to extract the evaluation point (s^*, t^*). Unlike for square root, for example, the evaluation point used for division can be encoded exactly in k bits. Let S and T be the k-bit unsigned integer encodings of s^* and t^*, respectively. The unsigned integer T can be extracted as presented in Chapter 2 (see Listing 2.1). Here we thus discuss only the computation of the integer S that encodes s^* as

$$S = s^* \cdot 2^{k-2}.$$

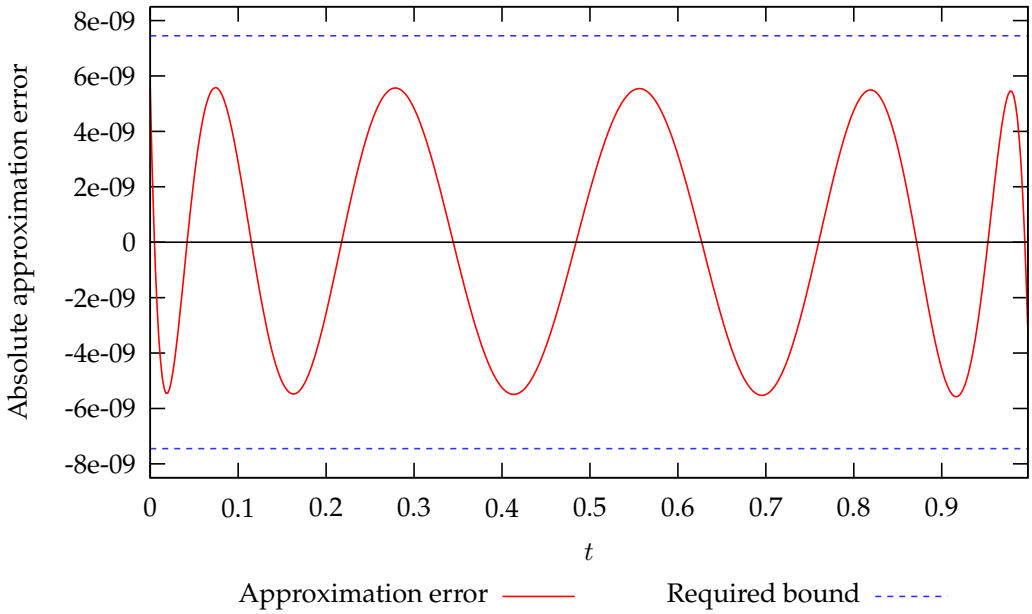


Figure 4.2: Absolute approximation error of $a(t)$ with respect to $1/(1+t)$ over $[0, 1-2^{-23}]$.

By definition $s^* = 2^{1-c} \cdot m'_x$. Assuming that the unsigned integer M_{px} encodes m'_x such as $M_{px} = m'_x \cdot 2^{k-1}$, we obtain the following:

$$S = M_{px} \cdot 2^{-c}.$$

In integer arithmetic, the computation of S may be done by shifting right M_{px} by $c \in \{0, 1\}$ bits. Since $c < 32$, the shift operation remains well defined in the sense of the C standard [Int99]. Hence, for the computation of the unsigned integer S may be done using the following piece of C code.

```
c = Mpx >= Mpy;

S = Mpx >> c;
```

Efficient polynomial evaluation and validation

Once we have the polynomial approximant $a(t)$, it remains to find an efficient evaluation program to evaluate $a(t)$ in fixed-point arithmetic and finite precision, so that the evaluation error satisfies (4.21). This can be done using CGPE (more details are given in Part II). Remark that as for square root, once the evaluation program is known (Listing 4.7 below), it remains finally to check:

- if no overflow occurs during the evaluation, that is, if each variable r_i fits in 32 bits,
- and if the evaluation error of the program is strictly less than the evaluation error bound η defined in (4.21).

Checking if no overflow occurs is done using Gappa, as presented in Section 5.1.1 (paragraph "Evaluation program validation"). However, as we have already said, the validation cannot be done directly since the sufficient condition on evaluation error cannot be satisfied on the whole input interval. Therefore, we have implemented a validation using a splitting by dichotomy of the input interval. This strategy is presented in the following section.

```
1  uint32_t __division_eval__ (uint32_t S, uint32_t T)
2  {
3    uint32_t r0  = mul(T, 0xfffe7d7);     // 0.32
4    uint32_t r1  = 0xffffffe8 - r0;       // 0.32
5    uint32_t r2  = mul(S, r1);            // 2.30
6    uint32_t r3  = 0x00000020 + r2;       // 2.30
7    uint32_t r4  = mul(T, T);             // 0.32
8    uint32_t r5  = mul(S, r4);            // 2.30
9    uint32_t r6  = mul(T, 0xffbad86f);    // 0.32
10   uint32_t r7  = 0xfffbece7 - r6;       // 0.32
11   uint32_t r8  = mul(r5, r7);           // 2.30
12   uint32_t r9  = r3 + r8;               // 2.30
13   uint32_t r10 = mul(r4, r5);           // 2.30
14   uint32_t r11 = mul(T, 0xf3672b51);    // 0.32
15   uint32_t r12 = 0xfd9d3a3e - r11;      // 0.32
16   uint32_t r13 = mul(T, 0x9a3c4390);    // 0.32
17   uint32_t r14 = 0xd4d2ce9b - r13;      // 0.32
18   uint32_t r15 = mul(r4, r14);          // 0.32
19   uint32_t r16 = r12 + r15;             // 0.32
20   uint32_t r17 = mul(r10, r16);         // 2.30
21   uint32_t r18 = r9 + r17;              // 2.30
22   uint32_t r19 = mul(r4, r4);           // 0.32
23   uint32_t r20 = mul(T, 0x1bba92b3);    // 0.32
24   uint32_t r21 = 0x525a1a8b - r20;      // 0.32
25   uint32_t r22 = mul(r4, 0x0452b1bf);   // 0.32
26   uint32_t r23 = r21 + r22;             // 0.32
27   uint32_t r24 = mul(r19, r23);         // 0.32
28   uint32_t r25 = mul(r10, r24);         // 2.30
29   uint32_t r26 = r18 + r25;             // 2.30
30 }
```

Listing 4.7: Evaluation program used for our *binary32* division.

4.5.3 Validation using a dichotomy-based strategy

From now, we have the evaluation program $\mathcal{P}$ and we have checked that no overflow occurs during the evaluation. Hence it remains to validate this program, that means to check if the evaluation error entailed by the evaluation of the polynomial P by the program $\mathcal{P}$ satisfies the bound in (4.21).

Subdomain-based error conditions

We have done this validation using Gappa. From (4.21) and the computed certified approximation error bound $\theta = 3 \cdot 2^{-29} \approx 2^{-27.41}$, we deduce that the evaluation error has to be strictly less than

$$\begin{aligned} \eta &= 2^{-25} - (4 - 2^{-21}) \cdot \theta \\ &\approx 2^{-26.9999}. \end{aligned} \tag{4.22}$$

When $m'_x \geq m'_y$, we can check with Gappa that the condition (4.21) is satisfied. However, when $m'_x < m'_y$, we can find some input (s^*, t^*) for which the condition (4.21) is not satisfied. For example, using GMP [Gra], we have considered all input (s^*, t^*), and for each of them, we have computed the result of the program of Listing 4.7 above in fixed-point arithmetic, and the exact rational result using GMP. Hence, using this approach we have found the following input

$$s^* = 3.935581684112548828125 \quad \text{and} \quad t^* = 0.9749044179916381835937 5,$$

and that for this input the evaluation error is bounded by $1074596671 \cdot 2^{-57} \approx 2^{-26.9988}$, which is slightly larger than the required bound (4.22).

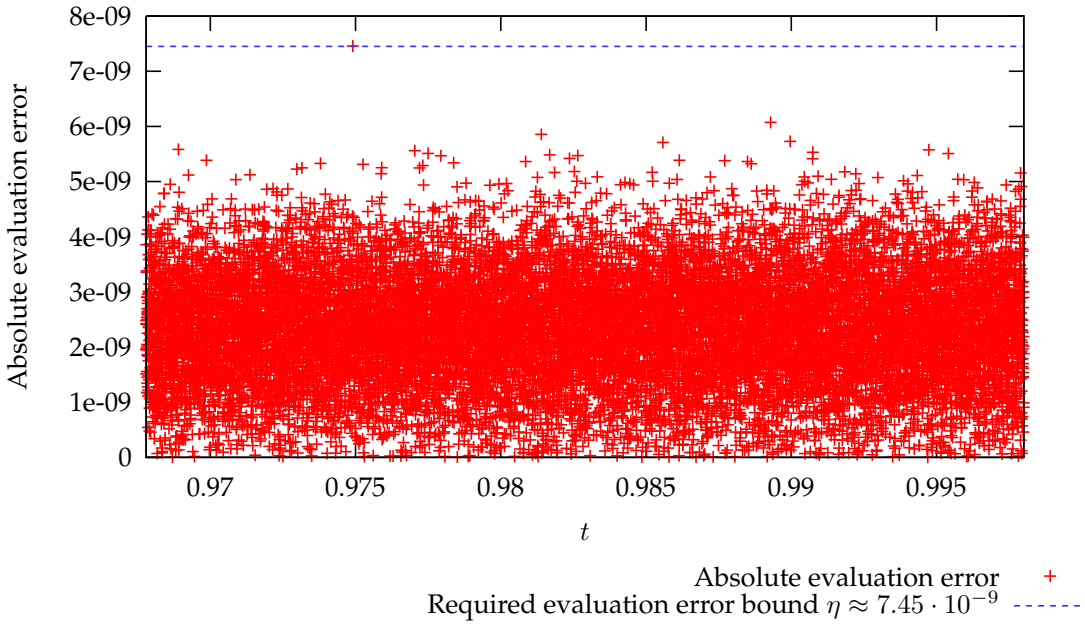


Figure 4.3: Absolute evaluation error $\rho(\mathcal{P})$, for $t \geq 0.9749044179916381835937 5$.

However, we know from Chebychev's theorem that the *approximation error* of a degree-n *minimax* polynomial a^* oscillates, so that the largest approximation error is reached at least $n+2$ times and the sign of this error alternates [Mul06, Theorem 7,§ 3.2,p. 32]. In our case, the *approximation error* $\alpha(a)$ of the degree-10 polynomial a with respect to function $1/(1+t)$ over $\mathcal{T}$ oscillates along the input interval, so that the maximum error is reached 12 times, as shown in Figure 4.2,

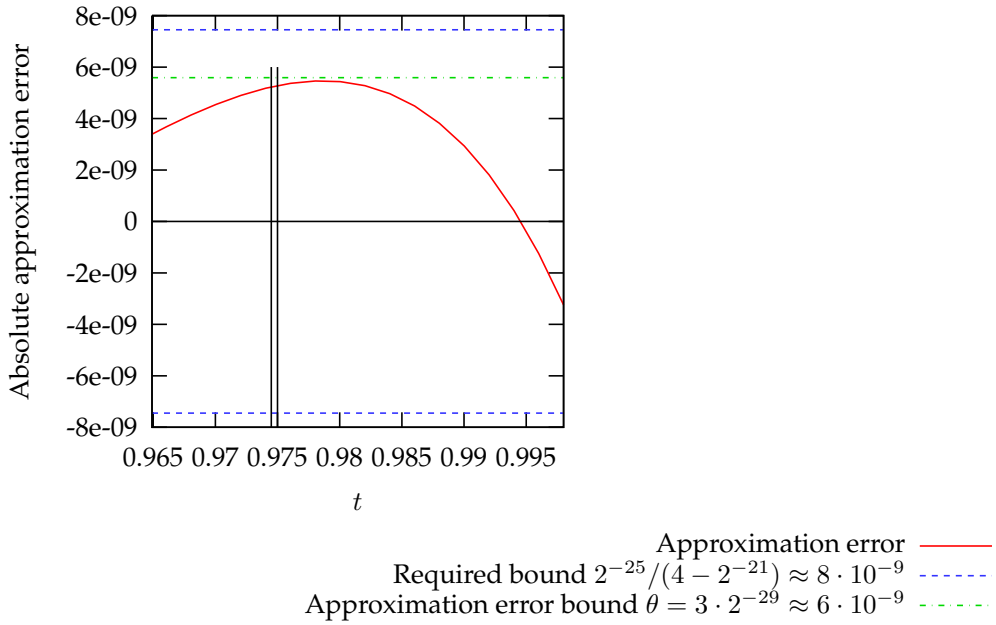


Figure 4.4: Absolute approximation error of $a(t)$ with respect to $1/(1+t)$, around $t = 0.9749044179916381835937 5$.

and may be smaller on some points than on the whole interval. Indeed, in Figure 4.4, we observe that if we consider a small interval around the point $t^* = 0.9749044179916381835937 5$, the *approximation error* of $a(t)$ on this small interval with respect to $1/(1+t)$ is slightly smaller than the one on the whole interval.

More particularly, let θ' be the approximation error of the polynomial approximant a with respect to $1/(1+t)$ at $t^* = 0.9749044179916381835937 5$:

$$\theta' = 1232560802107064287628542795321576594934 59 \cdot 2^{-164} \approx 2^{-27.4992},$$

and thus, the evaluation error on this particular input has to be less than

$$\eta' = 427554820082809494938604083452868552125485921363 \cdot 2^{-185} \approx 2^{-26.7732}.$$

From (4.22), we know that the evaluation error has to be strictly less than $2^{-26.9999}$. Hence, we can observe that for this particular input t^*, the evaluation error is slightly smaller than the required bound, and the condition (4.21) holds. The problem is that we cannot find "by hand" all the points for which the condition (4.21) is not satisfied and then checking the condition on these particular points.

Therefore, the method we propose (and which has been presented in [JKM+09]) consists in considering the input interval $\mathcal{I}$ as a union of n subintervals: $\mathcal{I} = \bigcup_{i=1}^{n} \mathcal{I}^{(i)}$, and defining the approximation and evaluation errors in (4.18) on each subinterval, such as

$$\alpha^{(i)}(a) = \max_{t \in \mathcal{I}^{(i)}} |1/(1+t) - a(t)| \quad \text{and} \quad \rho(\mathcal{P})^{(i)} = \max_{(s,t) \in \mathcal{S} \times \mathcal{I}^{(i)}} |P(s,t) - \mathcal{P}(s,t)|, \quad 1 \leq i \leq n. \tag{4.23}$$

Finally, if for $1 \leq i \leq n$ the approximation and evaluation errors satisfy

$$(4 - 2^{3-p}) \cdot \alpha^{(i)}(a) + \rho^{(i)}(\mathcal{P}) < 2^{-p-1}, \tag{4.24}$$

then (4.19) holds. It remains now to see how to find such a splitting into n subintervals.

Splitting interval by dichotomy

To find a splitting of the input domain $\mathcal{T}$ into n subintervals $\mathcal{T}^{(i)}$, so that the condition in (4.24) is satisfied on each subintervals, we have implemented a dichotomy search. More particularly, starting with $n = 1$ and $\mathcal{T}^{(i)} = \mathcal{T}$, we proceed as follows. For each subinterval $\mathcal{T}^{(i)}$:

1. first, we compute the approximation error bound $\theta^{(i)}$ of the polynomial approximant $a(t)$ with respect to $1/(1+t)$ over $\mathcal{T}^{(i)}$, such that
$$\alpha^{(i)}(a) \leq \theta^{(i)} \quad \text{and} \quad \theta^{(i)} < 2^{-25}/(4 - 2^{-21}),$$
2. then, we check if the evaluation error $\rho(\mathcal{P})^{(i)}$ over $\mathcal{T}^{(i)}$ satisfies
$$\rho(\mathcal{P})^{(i)} < \eta^{(i)} \quad \text{with} \quad \eta^{(i)} = 2^{-25} - (4 - 2^{-21}) \cdot \theta^{(i)},$$
and thus (4.24) holds.
3. and, if the condition in (4.24) is not satisfied, we split up the interval $\mathcal{T}^{(i)}$ into two subintervals, and launch this process on each subinterval.

Table 4.6 below illustrates this search by dichotomy. In the last column, "no" means that the considered interval in the second column has to be split up. The bounds on $\alpha^{(\cdot)}(a)$ and $\rho^{(\cdot)}(\mathcal{P})$ (where the $\cdot$ stands for the index of the interval in the subdivision) that are found using Sollya and Gappa are given in the third and fourth columns. (For the exact values of the θ_j's and η_l's we refer to [JKM+08, Appendix B].) This process has been launched in a 64-processor grid, and has found

Depth	Subintervals	$\alpha^{(\cdot)}(a) \leq$	$\rho^{(\cdot)}(\mathcal{P}) <$	Does (4.24) hold?
1	$I_{1,1} = [2^{-23}, 1 - 2^{-23}]$	$\theta_1 \approx 2^{-27.41}$	$\eta_1 \approx 2^{-26.99}$	no
2	$I_{2,1} = [2^{-23}, 0.5 - 2^{-23}]$	$\theta_2 \approx 2^{-27.41}$	$\eta_2 \approx 2^{-26.99}$	yes
	$I_{2,2} = [0.5, 1 - 2^{-23}]$	$\theta_1 \approx 2^{-27.41}$	$\eta_1 \approx 2^{-26.99}$	no
...				
j	$I_{j,1} = [2^{-23}, 0.5 - 2^{-23}]$	$\theta_2 \approx 2^{-27.41}$	$\eta_2 \approx 2^{-26.99}$	yes
	$I_{j,2} = [0.5, 0.75 - 2^{-23}]$	$\theta_1 \approx 2^{-27.41}$	$\eta_1 \approx 2^{-26.99}$	yes
	$I_{j,19309} = [0.921875, 0.9257811307907104492187 5]$	$\theta_3 \approx 2^{-27.44}$	$\eta_3 \approx 2^{-26.90}$	yes
	$I_{j,19533} = [0.97490406036376953125, 0.9749044179916381835937 5]$	$\theta_4 \approx 2^{-27.49}$	$\eta_4 \approx 2^{-26.77}$	yes

Table 4.6: Splitting steps.

a split up of the input domain $\mathcal{T}$ into $n = 36127$ subintervals, in about 5 hours.

4.6 Rounding condition implementation

This section explains how to implement, for each rounding-direction attribute, the rounding condition introduced in Section 2.2.4 in integer arithmetic. But before detailing the implementation of these conditions, let us give some general definitions, and some properties useful for rounding.

4.6.1 General definitions

Let $v = (01.v_1v_2 \ldots v_{k-2})_2$ be a value having a finite binary expansion in $k-2$ fraction bits that approximates the exact value ℓ from above, and obtained by polynomial evaluation (as described in Section 4.5), such that
$$v = \mathcal{P}(s^*, t^*) \quad \text{and} \quad -2^{-p} < \ell - v \leq 0.$$

Given this value v and given λ_r as in (4.5), to achieve correct rounding as detailed in Section 2.2.4, we need to deduce from the value v a value u such that

$$u = (u_0.u_1u_1 \dots u_p)_2 \quad \text{and} \quad |\ell \cdot 2^{-\lambda_r} - u| < 2^{-p}. \tag{4.25}$$

Hence, to obtain u as defined in (4.25), it suffices to truncate the value $v \cdot 2^{-\lambda_r}$ after p fraction bits. More particularly, by definition of the truncation function, we have $0 \leq v \cdot 2^{-\lambda_r} - u < 2^{-p}$. Also, we know from (4.6) that $\lambda_r \geq 0$. Then it follows that $-2^{-p} < \ell \cdot 2^{-\lambda_r} - v \cdot 2^{-\lambda_r} \leq 0$, since $-2^{-p} \leq -2^{-p-\lambda_r}$, and finally $|\ell \cdot 2^{-\lambda_r} - u| < 2^{-p}$.

Let V and U be the k-bit unsigned integers encoding of the computed value v and u, respectively, such as

$$V = v \cdot 2^{k-2} \quad \text{and} \quad U = u \cdot 2^{k-2}.$$

By definition of truncation in (2.7), we know that

$$\begin{aligned} U &= \lfloor (v \cdot 2^{2-k})/2^{p-\lambda_r} \rfloor \cdot 2^{-p} \cdot 2^{k-2} \\ &= \lfloor V/2^{k-p-2+\lambda_r} \rfloor \cdot 2^{k-p-2}. \end{aligned} \tag{4.26}$$

Let $K = k-p-2$. For a given format, the value K is a constant. Thus $U = \lfloor V/2^{K+\lambda_r} \rfloor \cdot 2^K$ can be obtained by shifting V right by λ_r bits and then by removing the K rightmost bits of the resulting integer (by taking the bitwise AND with $2^k - 2^K$). For example, for the *binary32* floating-point format, if follows from $(k, p) = (32, 24)$ that $K = 6$, so that

$$\begin{aligned} U &= \lfloor V/2^{6+\lambda_r} \rfloor \cdot 2^6 \\ &= (V \gg \lambda_r) \wedge (2^k - 2^6). \end{aligned} \tag{4.27}$$

Recall from (4.6) that λ_r belongs to $\{0, \dots, e_{\min} - p + 1\}$. Therefore, λ_r can be larger than $k-1$, a case where the behavior of the shift operator $\gg$ may be *not* specified (for example in C) [Int99]. To handle this case, notice first that $\lambda_r > p$ implies $v \cdot 2^{p-\lambda_r} \leq v \cdot 2^{-1} < 1$ since $v < 2$. Consequently u is zero and its encoding U is zero as well. Therefore, since $k > p + 2$, one way of constructing U from V when $\lambda_r > p$ is to shift V right by $\min(\lambda_r, p+1)$ instead of λ_r. And from (4.26), we get:

$$U = \Big(V \gg \min(\lambda_r, p+1)\Big) \wedge (2^k - 2^{k-p-2}),$$

more particularly on our *binary32* example, (4.27) thus becomes:

$$U = \Big(V \gg \min(\lambda_r, 25)\Big) \wedge (2^{32} - 2^6).$$

Its implementation in C is straighforward:

```
U = (V >> minu(Lr , 25)) & 0xFFFFFFC0;
```

4.6.2 Properties useful for correct rounding

In this section, we give some properties useful for implementing the correct rounding from the value u in 4.25. More particularly, we will determine if the exact result $r = x/y$ can be exactly a floating-point number, or halfway between two floating-point numbers.

The exact result of division can be a floating-point number

For division, Example 4.2 below shows that the exact value ℓ can be exactly a floating-point number.

Example 4.2. *Assume* $(k, p, e_{\max}) = (32, 24, 127)$. *Let* x *and* y *be two binary floating-point numbers such that*

$$m'_x = 1.01110001111111100001010_2 \quad \textit{and} \quad m'_y = 1.00010101110001000000000_2.$$

Since $m'_x \geq m'_y$, *we have* $\ell = m'_x/m'_x$ *and*

$$\ell = 1.01010101000000000000000_2,$$

that is, a significand having exactly $p-1$ *fraction bits.*

Hence, we conclude that the exact result $r = x/y$ defined in (4.3) can also be exactly a floating-point number.

The exact result of division can be a midpoint only if underflow occurs

Concerning midpoints, Property 4.3 below shows that the exact value ℓ cannot be halfway between two floating-point numbers.

Property 4.3. *For* x *and* y *two nonzero positive (sub)normal floating-point numbers defined in* (1.5) *in Section 1.2.1, the real value* ℓ *cannot be exactly halfway between two floating-point numbers.*

Proof. Assume first that $m'_x \geq m'_y$ and that ℓ is exactly halfway between two floating-point numbers. Then $c = 1$ and $m'_x = m'_y \cdot \ell$ with $\ell = \ell_0.\ell_1 \ldots \ell_{p-1}1$. By multiplying by 2^{2p-1}, we obtain

$$m'_x \cdot 2^{2p-1} = m'_y \cdot 2^{p-1} \cdot \underbrace{\ell_0\ell_1 \ldots \ell_{p-1}1}_{\text{odd integer}}.$$

Now let $q \in \mathbb{N}$ be such that $m_y \cdot 2^{p-1-q}$ is an odd integer. It follows that

$$m'_x \cdot 2^{2p-1-q} = \underbrace{m'_y \cdot 2^{p-1-q}}_{\text{odd integer}} \cdot \underbrace{\ell_0\ell_1 \ldots \ell_{p-1}1}_{\text{odd integer}}.$$

Since the fraction of m'_y has at most $p-1$ nonzero bits, we have $0 \leq q \leq p-1$ and $2 \leq 2^{p-q}$. Thus $m'_x \cdot 2^{2p-1-q}$ is an even integer and cannot be the product of two odd integers. Hence ℓ cannot be of the form $\ell = \ell_0.\ell_1 \ldots \ell_{p-1}1$. We can proceed similarly when $m_x < m_y$, which ends the proof. □

Thus, if no underflow occurs, that means that $\lambda_r = 0$, then the exact result $r = \ell \cdot 2^d$, as in (4.3), cannot be exactly halfway between two floating-point numbers. But, when an underflow occurs ($\lambda_r \geq 1$), since ℓ can be exactly a floating-point number, $\ell \cdot 2^{-\lambda_r}$, and the exact result as well, can be halfway between two floating-point numbers, as shown in Example 4.3.

Example 4.3. *Assume* $(k, p, e_{\max}) = (32, 24, 127)$. *Let* x *and* y *be two binary floating-point numbers defined in Example 4.2. We have*

$$m'_x = 1.01110001111111110000101_2, \quad m'_y = 1.00010101110001000000000_2,$$

and thus

$$\ell = 1.01010101000000000000000_2.$$

If $\lambda_r = 16$, *we conclude that*

$$\ell \cdot 2^{-\lambda_r} = 0.00000000000000001010101_2,$$

that is, exactly halfway between two floating-point numbers.

This property makes the rounding algorithm simpler when subnormal numbers are not supported (as in [JKM+09]), since we do not have to detect when ℓ is exactly halfway between two floating-point numbers. However, in our case (support of subnormal numbers), the exact result can be halfway between two floating-point numbers (when it lies in the range of subnormal numbers), which makes the rounding procedure more complicated, as we will see in Section 4.6.3 below.

4.6.3 How to implement the rounding condition?

In Section 4.6.1, we have seen how to compute the value u in (4.25) as the truncation of the approximation v. To be able to compute the correctly-rounded result, it remains now to implement the rounding condition cond introduced in Section 2.2.4. This condition depends on the rounding-direction attribute, but in all cases, it essentially relies on the ability to compare u to $\ell \cdot 2^{-\lambda_r}$. Let us give some elements on how to implement this rounding condition. Recall that $\ell = s/m'_y$ with $s = 2^{1-c} m'_x$. Also let U, M_{py}, and S be the k-bit unsigned integer encodings of u, m'_y, and s, respectively:

$$U = u \cdot 2^{k-2}, \quad M_{py} = m'_y \cdot 2^{k-1}, \quad S = s \cdot 2^{k-2}.$$

Note that the bit string of U is

$$U = \begin{cases} 0\cdots 0 & \text{if } \lambda_r > p, \\ \underbrace{00\cdots 00}_{\lambda_r+1} 1 v_1 \cdots v_{p-\lambda_r} \underbrace{00\cdots 00}_{k-p-2} & \text{if } \lambda_r \le p. \end{cases}$$

Now the goal consists in implementing in integer arithmetic the exact comparison between u and $\ell \cdot 2^{-\lambda_r}$. However, as for square root or reciprocal (see Chapter 3), the problem is that the value ℓ is not known exactly, and this comparison cannot be implemented directly. It turns out that the comparisons between $\ell \cdot 2^{-\lambda_r}$ and u may be implemented exactly by comparing $u \cdot m'_y$ and $s \cdot 2^{-\lambda_r}$, which have both a finite number of bits, instead of u and $\ell \cdot 2^{-\lambda_r}$. Since we want an implementation involving k-bit integers only, we aim further at reducing the comparison between $\ell \cdot 2^{-\lambda_r}$ and u to a comparison between k-bit integers. This is what the property below shows (for $\ge$, but the same holds for $>$).

Property 4.4. *Let $N = U \cdot 2^{-\lambda_r}$ and $Q = S/2$. Then N, Q are k-bit unsigned integers, that is, integers in $[0, 2^k - 1]$. Furthermore,*

$$u \ge \ell \cdot 2^{-\lambda_r} \quad \textit{if and only if} \quad N \cdot M_{py} \ge Q \cdot 2^k. \tag{4.28}$$

Proof. First, it follows from $k > p$ and $c \in \{0, 1\}$ that Q is a k-bit unsigned integer in $[0, 2^k - 1]$.

Now, for N, recall that $|\ell \cdot 2^{-\lambda_r} - u| < 2^{-p}$. Thus it follows that $u \cdot 2^{\lambda_r} < \ell + 2^{\lambda_r - p}$, and by multiplying by 2^{k-2}, we get

$$N < \ell \cdot 2^{k-2} + 2^{\lambda_r + k - p - 2}. \tag{4.29}$$

If $\lambda_r > p$ then $U = 0 = N$. Otherwise if $\lambda_r \le p$ then it follows from (4.29) and $\ell \in [1, 2)$ (see Property 4.1 above) that $N < 2^{k-1} + 2^{k-2}$. And in both cases, N is an integer in $[0, 2^k - 1]$. The equivalence in (4.28) is an immediate consequence of the definition of U, M_{py}, S, N, and Q. □

It follows from Property 4.4 that we could thus compare u and $\ell \cdot 2^{-\lambda_r}$ by computing the $2k$-bit product $N \cdot M_{py}$ exactly and compare it to the integer $Q \cdot 2^k$.

However, we will see that, in the same way as for square root and reciprocal in Chapter 3, for each rounding-direction attribute, it suffices to work with the higher "half" of $N \cdot M_{py}$. To do so, let defined the k-bit unsigned integer P as

$$P = \lfloor N \cdot M_{py} \cdot 2^{-k} \rfloor, \tag{4.30}$$

which satisfies

$$N \cdot M_{py} \cdot 2^{-k} - 1 < P \leq N \cdot M_{py} \cdot 2^{-k} \tag{4.31}$$

by definition of the floor function.

Let us now see how to implement the computation of P. The integer N may be computed by shifting U right by $\min(\lambda_r, p+1)$ bits. Moreover, by definition of the function `mul` as the floor function, the computation of P can be done as follows:

$$P = \texttt{mul}(N, M_{py}) \quad \text{with} \quad N = U \ll \min(\lambda_r, p+1).$$

For $(k, p, e_{\max}) = (32, 24, 127)$, the computation of N, P, and Q may be done using the following piece of C code.

```
N = U << minu(Lr , 25);

P = mul(N , Mpy);

Q = S >> 1;
```

From this statement and the definition above, we will see now for each rounding-direction attribute how to implement the rounding condition `cond` introduced in Section 2.2.4.

Implementation of the rounding condition for RoundTiesToEven

Recall that for RoundTiesToEven, the rounding condition is

$$\texttt{cond} = c_1 \vee c_2,$$

where

$$c_1 = u > \ell \cdot 2^{-\lambda_r} \quad \text{and} \quad c_2 = \big(u = \ell \cdot 2^{-\lambda_r} \text{ and } u_{p-1} = 0\big).$$

Moreover, in [JKMR08] for square root or [JKM+09] for division without subnormal numbers support, the implementation of RoundTiesToEven is quite simple since the exact result cannot be the middle of two floating-point numbers. On the contrary, here we do support subnormal numbers and thus we have to detect whether ℓ is the middle of two floating-point numbers or not. (This situation is handled by condition c_2 above.)

To do so and to implement the rounding condition, let us define the integer $Q' \in \{0, 1\}$ as follows

$$Q' = \begin{cases} 1 & \text{if } \text{“}2^k \text{ divides } N \cdot M_{py}\text{“} \quad \text{and} \quad u_{p-1} = 1, \\ 0 & \text{otherwise.} \end{cases}$$

By definition of $U = u \cdot 2^{k-2}$, we know that extracting the bits u_{p-1} may be done by taking the bitwise AND between U and 2^{k-1-p}. Then checking if this bit equals 1 may be done by comparing the resulting integer to 2^{k-1-p}. In our context and using the available ST231 instructions, for $(k, p, e_{\max}) = (32, 24, 127)$, the computation of Q' can thus be done as follows.

```
Qprime = ((N * Mpy) == 0x0) &  ((U & 0x80) == 0x80);
```

Here recall that `*` returns the 32 least significant bits of the 32×32-bit product.

Theorem 4.1. *The rounding condition $c_1 \vee c_2$ is equivalent to $P \geq Q + Q'$.*

Proof. Consider first $Q' = 1$ or $\big(Q' = 0$ and $u_{p-1} = 1\big)$. In this case the condition c_2 is false and thus the rounding condition is equivalent to $u > \ell \cdot 2^{-\lambda_r}$. Assume first $Q' = 1$. From Property 4.4, the inequality $u > \ell \cdot 2^{-\lambda_r}$ is equivalent to $N \cdot M_y > Q \cdot 2^k$. Since in this case 2^k divides $N \cdot M_y$,

then $u > \ell \cdot 2^{-\lambda_r}$ is also equivalent to $\lfloor N \cdot M_{py} \cdot 2^{-k} \rfloor > Q$, that is, by definition of P in (4.30), to $P \geq Q + 1 = Q + Q'$. Assume now $Q' = 0$. On the one hand, by Property 4.4 the inequality $u > \ell \cdot 2^{-\lambda_r}$ implies $N \cdot M_{py} > Q \cdot 2^k$ and since in this second case 2^k does not divide $N \cdot M_{py}$, it also implies $P + 1 > Q$, that is, $P \geq Q = Q + Q'$. On the other hand, using the right inequality of (4.31) together with the fact that 2^k does not divide $N \cdot M_{py}$, by definition the inequality $P \geq Q + Q'$ implies $N \cdot M_y > Q \cdot 2^k$ and also $u > \ell \cdot 2^{-\lambda_r}$.

Second, consider the case where $\big(Q' = 0$ and $u_p = 0\big)$. In this case the rounding condition is $u \geq \ell \cdot 2^{-\lambda_r}$. On the one hand, by Property 4.4 the condition $u \geq \ell \cdot 2^{-\lambda_r}$ implies $N \cdot M_y \geq Q \cdot 2^k$. Using the left inequality of (4.31), it follows that $u \geq \ell \cdot 2^{-\lambda_r}$ implies also $P + 1 > Q$ and thus $P \geq Q + Q'$. On the other hand, if $P \geq Q + Q'$ then $N \cdot M_{py} \cdot 2^{-k} \geq Q$ and thus $u \geq \ell \cdot 2^{-\lambda_r}$, which ends the proof. □

From Theorem 4.1, the condition $c_1 \vee c_2$ is equivalent to $P \geq Q + Q'$, that is, is true if and only if the C condition `P >= (Q + Qprime)` holds. For $(k, p, e_{\max}) = (32, 24, 127)$, the implementation of the rounding condition can be done as follows.

```
Qprime = ((N * Mpy) == 0x0) &  ((U & 0x80) == 0x80);

cond = (P >= Q + Qprime);
```

We know that S is a k-bit unsigned integer encoding of s, thus $S < 2^k$. Hence, since $Q = S/2$, the integer $Q + Q'$ is less than $2^{k-1} + 1$ and fits exactly in a k-bit unsigned integer.

Implementation of the rounding condition for RoundTowardPositive

As we have seen in Section 2.2.4, the difficulties for implementing the RoundTowardPositive algorithm come from the fact that these conditions depend on the sign of the result. Recall that the rounding tests are:

$$u \geq \ell \cdot 2^{-\lambda_r} \text{ if the result is positive,} \quad \text{and} \quad u > \ell \cdot 2^{-\lambda_r} \text{ if the result is negative.}$$

In order to implement them, let $Q' \in \{0, 1\}$ be defined as

$$Q' = \begin{cases} 1 & \text{if } \text{“}2^k \text{ divides } N \cdot M_{py}\text{“} \quad \text{and} \quad s_r = 1, \\ 0 & \text{otherwise.} \end{cases}$$

For $(k, p, e_{\max}) = (32, 24, 127)$, the computation of the integer Q' can be done using the following piece of C code.

```
Qprime = ((N * Mpy) == 0x0) & (Sr >> 31);
```

Theorem 4.2. *Assume that the result is positive ($s_r = 0$). The rounding condition $u \geq \ell \cdot 2^{-\lambda_r}$ is true if and only if $P \geq Q + Q'$.*

Proof. Here, $Q' = 0$ by assumption. Using (4.28), the inequality $u \geq \ell \cdot 2^{-\lambda_r}$ is equivalent to $N \cdot M_{py} \geq Q \cdot 2^k$. Then using the left inequality of (4.31), it follows that $P + 1 > Q$, that is, $P \geq Q = Q + Q'$. Conversely, using the right inequality of (4.31), we have $P \leq N \cdot M_y \cdot 2^{-k}$. Thus, if $P \geq Q$ then it follows that $N \cdot M_y \cdot 2^{-k} \geq Q$, that is, $u \geq \ell \cdot 2^{-\lambda_r}$, which ends the proof. □

Theorem 4.3. *Assume that the result is negative ($s_r = 1$). The rounding condition $u > \ell \cdot 2^{-\lambda_r}$ is true if and only if $P \geq Q + Q'$.*

Proof. Assume first $Q' = 1$. Then using (4.28), the inequality $u > \ell \cdot 2^{-\lambda_r}$ is equivalent to $N \cdot M_{py} > Q \cdot 2^k$. Since by definition of Q', 2^k divides $N \cdot M_{py}$, it follows that $u > \ell \cdot 2^{-\lambda_r}$ is equivalent to $P > Q$, that is, $P \geq Q + 1 = Q + Q'$. Consider now $Q' = 0$ and that 2^k does not divide $N \cdot M_{py}$. On the one hand, by definition, using (4.28), the inequality $u > \ell \cdot 2^{-\lambda_r}$ is equivalent to $N \cdot M_{py} > Q \cdot 2^k$. Using the left inequality of (4.31), we have $P + 1 > N \cdot M_{py} \cdot 2^{-k}$. It follows that, if $u > \ell \cdot 2^{-\lambda_r}$ then $P + 1 > Q$, that is, $P \geq Q = Q + Q'$. On the other hand, since 2^k does not divide $N \cdot M_{py}$, then the right inequality of (4.31) gives $P < N \cdot M_{py} \cdot 2^{-k}$. Thus, if $P \geq Q$ then $N \cdot M_{py} \cdot 2^{-k} > Q$, which is equivalent to $u > \ell \cdot 2^{-\lambda_r}$, that ends the proof. □

From Theorems 4.2 and 4.3, the rounding condition is equivalent to $P \geq Q + Q'$, that is, is true if and only if the C condition `P >= Q + Qprime` holds. We conclude that the implementation of rounding condition, for $(k, p, e_{\max}) = (32, 24, 127)$, may be done as follows.

```
Qprime = ((N * Mpy ) == 0x0) & (Sr >> 31);

cond = (P >= Q + Qprime);
```

Here again, we may check that the resulting integer $Q+Q' < 2^k$, and fits exactly in a k-bit unsigned integer.

Implementation of the rounding conditions for RoundTowardNegative

As we have seen in Section 2.2, the RoundTowardNegative algorithm depends also on the sign of the result, such that $\mathrm{RU}_p(-r) = -\mathrm{RD}_p(r)$. Thus, the implementation of this rounding algorithm can be easily deduced from the implementation of the RoundTowardPositive algorithm, and the proofs as well. To do so, let us first define the value $Q' \in \{0, 1\}$ as follows

$$Q' = \begin{cases} 1 & \text{if } \text{“}2^k \text{ divides } N \cdot M_{py}\text{“} \quad \text{and} \quad s_r = 0, \\ 0 & \text{otherwise.} \end{cases}$$

Using the same approach as what is done for RoundTowardPositive, the computation of Q' may be done with the following piece of C code.

```
Qprime = ((N * Mpy) == 0x0) & ((~Sr) >> 31);
```

From Theorems 4.2 and 4.3, we get the following corollary.

Corollary 4.1. *If the result is positive ($s_r = 0$) then the rounding condition $u \geq \ell \cdot 2^{-\lambda_r}$ is true if and only if $P \geq Q + Q'$. If the result is negative ($s_r = 1$) then the rounding condition $u > \ell \cdot 2^{-\lambda_r}$ is true if and only if $P \geq Q + Q'$.*

From Corollary 4.1, we deduce that the rounding condition is equivalent to $P \geq Q + Q'$: this condition is true if and only if the C condition `P >= Q + Qprime` holds. And the implementation of rounding condition, for $(k, p, e_{\max}) = (32, 24, 127)$, may be done as follows.

```
Qprime = ((N * Mpy) == 0x0) & ((~Sr) >> 31);

cond = (P >= Q + Qprime);
```

Implementation of the rounding condition $u > \ell \cdot 2^{-\lambda_r}$ for RoundTowardZero

The RoundTowardZero rounding-direction attribute can be seen as the simplest to be implemented, since it does not depend on the sign of the result and we do not have to detect if ℓ is

exactly the middle of two floating-point numbers: the rounding condition simply is $u > \ell \cdot 2^{-\lambda_r}$. To do so, let us define $Q' \in \{0, 1\}$ as

$$Q' = \begin{cases} 1 & \text{if } \text{“}2^k \text{ divides } N \cdot M_{py}\text{“}, \\ 0 & \text{otherwise.} \end{cases}$$

whose computation can be done as follows.

```
Qprime = ((N * Mpy) == 0x0);
```

From Theorem 4.3, we can derive the following corollary.

Corollary 4.2. *For RoundTowardZero, the rounding condition is true if and only if* $P \geq Q + Q'$.

We still may check that $Q + Q'$ fits exactly in k-bit integer since $Q + Q' < 2^k$. And from Corollary 4.2, the rounding condition is equivalent to $P \geq Q + Q'$: finally this condition is true if and only if the C condition `P >= Q + Qprime` holds. The rounding condition can be implemented as follows.

```
Qprime = ((N * Mpy) == 0x0);

cond = (P >= Q + Qprime);
```

PART II

Code generation for the efficient and certified evaluation of polynomials in fixed-point arithmetic

CHAPTER 5

Polynomial evaluation in fixed-point arithmetic on the ST231 processor

This chapter gives an overview of classical methods for evaluating a degree-n polynomial, and gives some examples of the evaluation of polynomials in fixed-point arithmetic on the ST231 processor. These methods are Horner's rule, "second-order Horner's rule", Estrin's method, and a method derived from Estrin's method and particularly well-adapted for evaluating our particular bivariate polynomial (Part I). This chapter points out the fact that these classical methods are accurate enough for implementing some operators in FLIP. However in term of evaluation latency, they remain less efficient than the best schemes found by using CGPE (Part II). Finally, through two examples, Knuth and Eve's algorithm and Paterson and Stockmeyer's algorithm, it explains why methods based on coefficient adaptation (which involve fewer multiplications than with Horner's rule) are not well-adapted for integer processors, like the ST231.

The implementation of mathematical functions may rely on the evaluation of an accurate enough polynomial approximant of the function on a reduced interval (see [DLDdD+], [Lau08], or [Ylö06] for examples). Indeed, we have shown in Part I that the implementation of several mathematical functions like roots and their reciprocals (square root, reciprocal square root, reciprocal, ...) and division may be reduced to the evaluation of a particular bivariate polynomial. And since these polynomials have to be evaluated at runtime, we have chosen to implement them using polynomials of smallest degree. But even if the degree is the smallest, the evaluation of this polynomial approximant remains the most expensive part of the whole codes of such functions. Hence we have to find ways to evaluate this polynomial that are as efficient as possible. The most naive method for evaluating a degree-n polynomial $a(x)$ defined in the monomial basis

$$a(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2 + \cdots + a_n \cdot x^n \quad \text{with} \quad a_n \neq 0,$$

consists in evaluating each monomial $a_i \cdot x^i$ and summing them together using n additions. This approach remains inefficient, since the evaluation involves a lot of calculations, especially a lot of multiplications (exactly $2n-1$, see [Knu98, §4.6.4]).

Other methods evaluate this polynomial $a(x)$ in a more efficient way. Horner's rule evaluates the polynomial $a(x)$ in n additions and n multiplications, that is, $2n$ operations, in a fully sequential way. Assuming an addition in 1 cycle and a multiplication in 3 cycles (as on the ST231 processor), the evaluation latency is of $4n$ cycles. On the contrary, Estrin's method tends to expose

instruction-level parallelism, but at the price of an increase of the total number of operations (n additions and about $n + \log(n+1)$ multiplications).

In the 60's, multiplication was much slower than addition.[1] This statement has motivated the implementation of evaluation schemes that reduce the number of multiplications. These schemes are based on a preliminary transformation of the polynomial, called *adaptation* or *preconditioning*. Actually this transformation may be costly, but is done once for all the evaluations. Among these methods based on coefficient adaptations, let us mention Knuth and Eve's algorithm [Knu98, §4.6.4, Theorem E], [Eve64] and Paterson and Stockmeyer's algorithm [PS73]. The former enables to evaluate a degree-n polynomial in n additions and about $n/2$ multiplications, while the latter uses $3n/2$ additions and about $n/2 + \log_2(n)$ multiplications. That is, for large values of n, both divide by roughly two the number of multiplications used.

More and more, the cost of multiplication is reduced, and tends to be equivalent to the one of the addition. In particular, since the `fma` is now required by the IEEE 754-2008 standard, the constructors may tend to implement multiplication and addition as special cases of the `fma` operation, and both these operations will have the same cost. This is actually already done on the Itanium® [CHT02, p. 69], for example. Hence, improving the evaluation of a polynomial will not rely on the reduction of the number of multiplications any more, especially if it involves an increase of the number of additions. And thus, we will prefer methods that expose as much instruction-level parallelism as possible.

Recall that in Part I, we have seen that the implementation of several mathematical functions, in binary floating-point arithmetic, can be made very efficient on ST231 thanks to the fast evaluation of particular bivariate polynomials of the form

$$P(s,t) = 2^{-p-1} + s \cdot (a_0 + a_1 \cdot t + \cdots + a_\delta \cdot t^\delta), \tag{5.1}$$

This chapter studies the evaluation of $P(s,t)$ in fixed-point arithmetic. It is organized as follows. Section 5.1 presents some classical methods usually used for evaluating univariate polynomials, especially for implementing mathematical functions, and explains how they can be extended for evaluating the polynomial in (5.1) above. Then Section 5.2 shows why methods based on coefficient adaptation are not well-suited to our context of fixed-point polynomial evaluation, especially on the ST231.

5.1 Classical polynomial evaluation schemes

This section presents some classical polynomial evaluation schemes, and explains how they can be used for evaluating efficiently the polynomial P in (5.1) in fixed-point arithmetic, on integer VLIW processor cores and in particular on the ST231. Most of these evaluation schemes are well-adapted for evaluating univariate polynomials. But they may also be extended for evaluating bivariate polynomials, and more particularly the polynomial P in (5.1).

More precisely, this section presents three classical evaluation schemes: Horner's rule, an extension called "second order" Horner's rule, and Estrin's method.

5.1.1 Horner's rule

Originally introduced in the seventeenth century by Newton, Horner's rule is the most-commonly used evaluation scheme for evaluating a univariate polynomial in floating-point arithmetic. For example, the evaluation of polynomials for implementing some mathematical functions in CR-Libm [DLDdD+] is done using this method (see also [Lau08]). In 1966 [Pan66], Pan showed that

[1] "Using floating-point hardware, a multiplication will take perhaps 5 times as long as an addition. In fixed-point arithmetic [...], a multiplication will often take up to 20 or more times as long as an addition." [Knu62].

Horner's rule for evaluating a degree-n polynomial with coefficients given in the monomial basis, as

$$a(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2 + \cdots + a_n \cdot x^n \quad \text{with} \quad a_n \neq 0,$$

minimizes the numbers of multiplications and additions [Knu98, p. 519, Exercise 38]. It evaluates the polynomial $a(x)$ using n additions and n multiplications as follows:

$$a(x) = a_0 + x \cdot \Big(a_1 + x \cdot \big(a_2 + \cdots + x \cdot (a_{n-1} + x \cdot a_n) \ldots\big)\Big).$$

This evaluation scheme is fully sequential. More particularly, on the ST231 processor, with an addition in 1 cycle and a multiplication in 3 cycles, this evaluation is done in theoretically n steps of 1 multiplication and 1 addition, that is, in $4n$ cycles.

In [Bol04], Boldo recalls that usually Horner's evaluation scheme has good numerical properties, especially when the intermediate x is not too close to a zero of the polynomial to be evaluated. This is actually the case for the polynomials we have used for implementing the functions of FLIP (see also [BD04] or [Hig02], for example). But since the evaluation is done in a fully sequential way, it may be ill-suited for 4-issue processors, like the ST231 processor.

Note also that some methods have been proposed to extend Horner's rule to the evaluation of multivariate polynomials [CK04], [PS00]. However, in our case, for evaluating our particular bivariate polynomial P in (5.1), we can simply evaluate the polynomial $a(x)$ with Horner, and then compute a last step (multiplication by s and addition of 2^{-p-1}), as follows:

$$P(s,t) = 2^{-p-1} + s \cdot \Big[a_0 + t \cdot \Big(a_1 + t \cdot \big(a_2 + \cdots + t \cdot (a_{n-1} + t \cdot a_n) \ldots\big)\Big)\Big].$$

Let us now see how to implement this evaluation scheme on an integer architecture, through an example taken from our code for square root.

First example of evaluation via Horner's rule

Assume that $(k, p, e_{\max}) = (32, 24, 127)$ and let us consider the polynomial P defined in Chapter 3 for the implementation of square root. This polynomial has the form given in (5.1), with $\delta = 8$:

$$P(s,t) = 2^{-25} + s \cdot \sum_{i=0}^{8} a_i \cdot t^i.$$

The values of the coefficients a_i are recalled in Table 5.1 below. The points $(\hat{s}, t)$ at which we have to evaluate P are as follows:

$$\hat{s} \in \{1, 1.01101010000010011110011001101002\} \quad \text{and} \quad t \in [0, 1 - 2^{-23}],$$

with t having 23 fraction bits. The evaluation of the polynomial $P(s,t)$ with Horner's rule is done as follows:

$$\begin{aligned} P(s,t) \quad = \quad & 2^{-25} + s \cdot \Big[a_0 + t \cdot \Big[a_1 + t \cdot \Big(a_2 + t \cdot \Big(a_3 + t \cdot \big(a_4 + \\ & t \cdot \big(a_5 + t \cdot (a_6 + t \cdot (a_7 + t \cdot a_8))\big)\big)\Big)\Big)\Big]\Big]. \end{aligned}$$

Recall that from Section 3.5.4, we know that the polynomial $a(t)$ has to approximate the function $(1+t)^{1/2}$ with an approximation error $\alpha(a) \leq \theta$, where $\theta < (2^{-25} - 2^{-31.5})/2^{1/2} \approx 2^{-25.51}$. In

Coefficient	Sign	Value	Encoding integer	Format
a_0	+	1.00000000325962901115417480468 75	`0x80000007`	Q_{31}
a_1	+	0.49999940209090709686279296875	`0x3ffffafc`	Q_{31}
a_2	-	0.12498275609686970710754394531 25	`0x0fff6f59`	Q_{31}
a_3	+	0.06230625417083501815795898437 5	`0x07f9a6be`	Q_{31}
a_4	-	0.03794703539460897445678710937 5	`0x04db72ce`	Q_{31}
a_5	+	0.02358471788465976715087890625	`0x0304d2f4`	Q_{31}
a_6	-	0.01247844425961375236511230468 75	`0x0198e4c7`	Q_{31}
a_7	+	0.00451590353623032569885253906 25	`0x0093fa25`	Q_{31}
a_8	-	0.00078448653221130371093 75	`0x0019b4c0`	Q_{31}

Table 5.1: Coefficients of the polynomial $a(t)$ used for our square root implementation.

our case, the polynomial $a(t)$ defined in Table 5.1 has an approximation error θ with respect to the function $(1+t)^{1/2}$ given by:

$$\begin{aligned} \theta &= 166306437400395567390185987137844671 \times 2^{-145} \\ &\approx 2^{-27.99}, \end{aligned} \tag{5.2}$$

which is actually strictly less than $(2^{-25} - 2^{-31.5})/2^{1/2} \approx 2^{-25.51}$, as required. Using (5.2), it follows from (3.37) in Section 3.2.2, Chapter 3, that we have to evaluate the polynomial $P(s,t)$, with an evaluation error $\rho(\mathcal{P}) \leq \eta$, with:

$$\begin{aligned} \eta &= 269837285807227497773782362822784117 \times 2^{-143} \\ &\approx 2^{-25.30}. \end{aligned} \tag{5.3}$$

When $(k, p, e_{\max}) = (32, 24, 127)$, the implementation of this scheme in fixed-point arithmetic may be done using the C code presented in Listing 5.1 below. Here, we can observe that since each intermediate variable r_i is signed, we use signed multiplication, emulated with the function `mul64h` presented in Appendix B, Section B.3. More particularly, each integer variable encodes a real value in signed fixed-point arithmetic presented in Section 1.2.3, with one bit for handling the sign, as explained in Remark 1.1.

However, from Remark 1.2, we know that multiplying two fixed-point numbers with one bit handling the sign each leads to a result with "two sign bits". Hence, on this small example, we observe that after each multiplication, we have to shift left the resulting integer to keep only one sign bit. Otherwise, the "sign part" would get larger after each multiplication, to the detriment of the fraction part, and we would lose accuracy. Indeed, the intermediate integers would then encode real values in formats having smaller and smaller fraction parts. Recalling that on the ST231 processor, additions, subtractions, and shifts all have a latency of 1 cycle, while multiplications have a latency of 3 cycles. Hence, using Horner's rule presented in Listing 5.1, and assuming only addition, subtraction, shift, and multiplication, the degree-9 polynomial P can be evaluated in 45 cycles on an integer architecture, instead of the $9 \times (3+1) = 36$ cycles expected for Horner's rule in degree 9.

However, on the ST231 processor core there is a specific instruction, named `shBadd`, with `B` $\in \{1, 2, 3, 4\}$, which computes `(X << B)+ Y` (with X and Y being two integers) in 1 cycle [ST208b, pp. 277-284]. For example, the lines 7 and 8 in Listing 5.1 yield the following piece of ST231 assembly code.

```
1  uint32_t __sqrt_eval__signed_horner__(uint32_t T, uint32_t S)
2  {
3    uint32_t Tsigned = T >> 1;                          //  0.32 -> s0.31
4
5    int32_t r0  = mul64h(Tsigned , 0x0019b4c0) << 1;    // s0.31
6    int32_t r1  = 0x0093fa25 - r0;                      // s0.31
7    int32_t r2  = mul64h(Tsigned , r1) << 1;            // s0.31
8    int32_t r3  = r2 - 0x0198e4c7;                      // s0.31
9    int32_t r4  = mul64h(Tsigned , r3) << 1;            // s0.31
10   int32_t r5  = r4 + 0x0304d2f4;                      // s0.31
11   int32_t r6  = mul64h(Tsigned , r5) << 1;            // s0.31
12   int32_t r7  = r6 - 0x04db72ce;                      // s0.31
13   int32_t r8  = mul64h(Tsigned , r7) << 1;            // s0.31
14   int32_t r9  = r8 + 0x07f9a6be;                      // s0.31
15   int32_t r10 = mul64h(Tsigned , r9) << 1;            // s0.31
16   int32_t r11 = r10 - 0x0fff6f59;                     // s0.31
17   int32_t r12 = mul64h(Tsigned , r11) << 1;           // s0.31
18   int32_t r13 = r12 + 0x3ffffafc;                     // s0.31
19   int32_t r14 = mul64h(Tsigned , r13) << 1;           // s0.31 -> 1.31
20   int32_t r15 = r14 + 0x80000007;                     //  1.31
21   int32_t r16 = mul(r15 , S);                         //  2.30
22   int32_t r17 = r16 + 0x20;                           //  2.30
23 }
```

Listing 5.1: Polynomial evaluation with Horner's rule in signed fixed-point arithmetic.

```
mul64hu $r18 = $r16, $r18       ## (cycle 6)
nop                             ## (cycle 6)
 ;; ## (bundle 4)
sh1add $r18 = $r18, -26797255   ## (cycle 9)
 ;; ## (bundle 5)
```

(Remark that, for example, the shift and addition instructions in lines 7 and 8 in Listing 5.1, respectively, are caught by the ST231 compiler, and the above ST231 assembly code is automatically generated.) Hence, using this specific instruction `sh1add` leads to an evaluation in 38 cycles on ST231 of the evaluation program in Listing 5.1. This is already more than 15 % faster than the previous code (45 cycles), but still almost 3 times slower than the 13-cycle evaluation code we are able to automatically generate and validate in this case (see Section 3.5.5 and paragraph "Evaluation program validation" below).

We may check with Gappa [Mel], [Mel06] that the evaluation error $\rho(\mathcal{P})$ of the program in Listing 5.1 is bounded as follows:

$$\begin{aligned}\rho(\mathcal{P}) &\leq 479676712772028613 \times 2^{-86} \\ &\lesssim 2^{-27.2652},\end{aligned}$$

and thus is actually strictly less than the required bound $\eta \approx 2^{-25.30}$ in (5.3). (The way used for validating an evaluation program with Gappa is detailed in paragraph "Evaluation program validation" later in this section). That means that using Horner's rule in signed fixed-point arithmetic enables to ensure correct rounding of square root. However, we have observed that this scheme does not allow to get good performances in terms of latency, even if a specific instruction like `shBadd` is available.

Unsigned fixed-point evaluation and arithmetic operator choice

It follows from the previous paragraph that using signed fixed-point arithmetic may lead to a degradation of performances, more particularly an increase of the evaluation latency. Therefore, we will see now how to implement this evaluation scheme using only unsigned fixed-point arithmetic.

More precisely, we will see now how to implement this evaluation scheme, so that each computed intermediate variable r_i remains of constant sign, either always + or always -. To do so, let us consider an intermediate variable r_i. Three cases may occur, when the inputs s, t vary:

- If $r_i > 0$ then the variable is always positive: this is the simplest case, and no special handling has to be done;
- If $r_i < 0$ then the variable is always negative: in this case, the variable is stored in absolute value, and the handling of its sign is propagated further in the evaluation;
- Otherwise, the evaluation scheme cannot be implemented using only unsigned fixed-point arithmetic: either it is implemented using signed fixed-point arithmetic as presented above, or it is rejected and another one has to be chosen.

Using this approach, we may check with MPFI [CLN+] (see also [RR05]) or Gappa that each variable r_i lies in a positive or negative interval, and does not change of sign while the inputs vary. Finally the polynomial defined in Table 5.1 can be implemented using exclusively unsigned fixed-point arithmetic, that is, unsigned 32-bit integers, as presented in Listing 5.2 below. Therefore the

```
1  uint32_t __sqrt_eval__horner__(uint32_t T, uint32_t S)
2  {
3    uint32_t r0  = mul(T, 0x0019b4c0);     // 1.31
4    uint32_t r1  = 0x0093fa25 - r0;        // 1.31
5    uint32_t r2  = mul(T, r1);             // 1.31
6    uint32_t r3  = 0x0198e4c7 - r2;        // 1.31
7    uint32_t r4  = mul(T, r3);             // 1.31
8    uint32_t r5  = 0x0304d2f4 - r4;        // 1.31
9    uint32_t r6  = mul(T, r5);             // 1.31
10   uint32_t r7  = 0x04db72ce - r6;        // 1.31
11   uint32_t r8  = mul(T, r7);             // 1.31
12   uint32_t r9  = 0x07f9a6be - r8;        // 1.31
13   uint32_t r10 = mul(T, r9);             // 1.31
14   uint32_t r11 = 0x0fff6f59 - r10;       // 1.31
15   uint32_t r12 = mul(T, r11);            // 1.31
16   uint32_t r13 = 0x3ffffafc - r12;       // 1.31
17   uint32_t r14 = mul(T, r13);            // 1.31
18   uint32_t r15 = 0x80000007 + r14;       // 1.31
19   uint32_t r16 = mul(S, r15);            // 2.30
20   uint32_t r17 = 0x00000020 + r16;       // 2.30
21   return r17;
22 }
```

Listing 5.2: Polynomial evaluation with Horner's rule in unsigned fixed-point arithmetic.

evaluation of the program in Listing 5.2 may be done without any increase of latency, that is, in exactly 36 cycles, as expected for Horner's rule in degree 9 and with latencies of 1 and 3 cycles for, respectively, addition and multiplication. In addition to that statement, we can check with Gappa that the evaluation program $\mathcal{P}$ implemented in Listing 5.2 has an evaluation error $\rho(\mathcal{P})$

bounded as follows:

$$\begin{aligned}\rho(\mathcal{P}) &\leq 1103468601438365355 \times 2^{-88} \\ &\lesssim 2^{-28.0633},\end{aligned}$$

which, again, is strictly less than the required bound $\eta \approx 2^{-25.30}$ in (5.3). A possible explanation of the improvement compared to the evaluation error of Listing 5.1 ($\approx 2^{-27.2652}$) may be due to the fact that, in Listing 5.1, all the results of signed multiplications are in the format Q_{30} and then adjusted in the format Q_{31} by shifting 1 bit left. On the contrary, in Listing 5.2, the results of unsigned multiplications are already in the format Q_{31}, and thus we gain one bit, and it could explain why we have an evaluation error almost twice more accurate. That means that this evaluation program could be also used for implementing correctly-rounded square root in the *binary32* floating-point format.

Evaluation program validation

Once we have written the evaluation program as a piece of C code, we have seen that it remains to validate it, that is, to check if the error entailed by the evaluation of this program satisfies the evaluation error bound η in (5.3). Until now we have seen that the evaluation error using Horner's rule is less than the required bound, but we do not have seen how to determine, in practice, a bound on this evaluation error.

This can be done either using certified interval arithmetic with MPFI, or using Gappa. Here, we will briefly explain how to write a Gappa script for validating the evaluation program of Listing 5.2 above. With our Gappa script, we want to check:

- if no overflow occurs during polynomial evaluation, that is, if all the variables are no larger than $2^{32} - 1$, so that they fit exactly in 32-bit unsigned integers;
- if the program is accurate enough, that is, if its evaluation error $\rho(\mathcal{P})$ is no larger than the required bound η in (5.3).

Checking with Gappa that during the evaluation all the variables fit in 32-bit unsigned integers, that is, that no overflow occurs, is done as follows. Given an intermediate variable r_i in the Q_f format, we check with Gappa that the variable r_i lies in the range $[0, (2^{32} - 1)/2^f]$, with f the number of fraction bits of r_i. By definition of the function `mul`, this *overflow* case may occur only for additions. For example, assuming that the variables r_0 and r_1 are both in the Q_{31} format (Listing 5.2), this property can be checked using lines 33 and 34 of the piece of Gappa code in Listing 5.3 below.

Let us now see how to check if the evaluation error $\rho(\mathcal{P})$ is upper bounded by η in (5.3). In the Gappa script in Listing 5.3 below, the variable Mr_i represents the "ideal mathematical" value approximated by r_i. Assuming that the result of the evaluation is stored in r_{17}, checking if $|r_{17} - Mr_{17}| \leq \eta$ is done as follows:

$$|r_{17} - Mr_{17}| - \eta \leq 0.$$

Hence the evaluation error can be checked as shown in line 36, while a bound on this evaluation error can be computed using line 37 (Listing 5.3).

5.1.2 Extension to second-order Horner's rule

In this section, we will see how to extend Horner's rule in order to expose more instruction-level parallelism. Indeed, the main drawback of the Horner's rule relies on the fact that the evaluation

```
# Definition
cst = fixed<-30,dn>(cst0);                  # cst = 2^-25 in the format Q2.30
a0 = fixed<-31,dn>(coef0);                  # each coefficient a_i is in the
                                            #  format Q31
# ...

T = fixed<-32,dn>(fixed<-23,dn>(var0));     # T is in the format Q32
                                            #  with at most 23 fraction bits

S = fixed<-31,dn>(var1);                    # S is in the format Q1.31

CertifiedBoundEta = 269837285807227497773782362822784117b-143;

## Evaluation scheme
r0  fixed<-31,dn>= T * a8;       Mr0 = T * a8;      # r0 = T * a8 truncated
                                                    #  after 31 fraction bits
r1  fixed<-31,dn>= a7 - r0;      Mr1 = a7 - Mr0;
# ...
r15 fixed<-31,dn>= a0 + r14;     Mr15 = a0 + Mr14;
r16 fixed<-30,dn>= S * r15;      Mr16 = S * Mr15;
r17 fixed<-30,dn>= cst + r16;    Mr17 = cst + Mr16;

## Results
{
  (
       var0   in [0x00000000p-32,0xfffffe00p-32]
    /\ var1   in [0x80000000p-31,0xb504f334p-31]
    /\ cst0   in [0x00000020p-30,0x00000020p-30]
    /\ coef0  in [0x80000007p-31,0x80000007p-31]
    /\ coef1  in [0x3ffffafcp-31,0x3ffffafcp-31]
#...
    ->
       r0  in [0,0xffffffffp-31]  # does r0 lie in [0,(2^32-1)/2^31]?
    /\ r1  in [0,0xffffffffp-31]
#...
    /\ |r17 - Mr17| - CertifiedBoundEta <= 0
    /\ |r17 - Mr17| in ?
  )
}
```

Listing 5.3: Piece of Gappa code to validate polynomial evaluation.

is done in a fully sequential way, and is extremely ill-suited to be used on the ST231 processor, since it uses only one of the four issues available. A first improvement consists in "splitting up" the polynomial $a(x)$ into its odd and even parts, and in evaluating both separately using Horner's rule. On our example, this gives the following parenthesization:

$$\begin{aligned} P(s,t) &= \left[2^{-25} + (s \cdot t) \cdot \left(a_1 + t^2 \cdot \left(a_3 + t^2 \cdot (a_5 + t^2 \cdot a_7)\right)\right)\right] + \\ &\quad s \cdot \left[a_0 + t^2 \cdot \left(a_2 + t^2 \cdot \left(a_4 + t^2 \cdot (a_6 + t^2 \cdot a_8)\right)\right)\right]. \end{aligned}$$

Roughly, this scheme, called *second-order Horner's rule* [Knu98, §4.6.4], allows to evaluate two polynomials of degree half the initial degree in parallel, and to combine them together using one step of Horner's rule. Assuming an addition in 1 cycle, a multiplication in 3 cycles, and unbounded parallelism, the critical path of evaluation consists of:

- the computation of t^2 (3 cycles),
- the evaluation of the two polynomials of degree 4 via Horner's rule (16 cycles),
- and one step of Horner's rule for combining both subpolynomials (4 cycles).

Hence a total expected latency of 23 cycles. We may observe that, not surprisingly, the implementation of this evaluation scheme on the ST231 processor indeed uses two issues as illustrated on Figure 5.1.

```
1  uint32_t __sqrt_eval__2nd_horner__(uint32_t T, uint32_t S)
2  {
3    uint32_t r0  = mul(S, T);              // 1.31
4    uint32_t r1  = mul(T, T);              // 0.32
5    uint32_t r2  = mul(r1, 0x0093fa25);    // 1.31
6    uint32_t r3  = 0x0304d2f4 + r2;        // 1.31
7    uint32_t r4  = mul(r1, r3);            // 1.31
8    uint32_t r5  = 0x07f9a6be + r4;        // 1.31
9    uint32_t r6  = mul(r1, r5);            // 1.31
10   uint32_t r7  = 0x3ffffafc + r6;        // 1.31
11   uint32_t r8  = mul(r0, r7);            // 2.30
12   uint32_t r9  = 0x00000020 + r8;        // 2.30
13   uint32_t r10 = mul(r1, 0x0019b4c0);    // 1.31
14   uint32_t r11 = 0x0198e4c7 + r10;       // 1.31
15   uint32_t r12 = mul(r1, r11);           // 1.31
16   uint32_t r13 = 0x04db72ce + r12;       // 1.31
17   uint32_t r14 = mul(r1, r13);           // 1.31
18   uint32_t r15 = 0x0fff6f59 + r14;       // 1.31
19   uint32_t r16 = mul(r1, r15);           // 1.31
20   uint32_t r17 = 0x80000007 - r16;       // 1.31
21   uint32_t r18 = mul(S, r17);            // 2.30
22   uint32_t r19 = r9 + r18;               // 2.30
23   return r19;
24 }
```

Listing 5.4: Polynomial evaluation by means of second-order Horner's rule in unsigned fixed-point arithmetic.

In terms of latency of evaluation using second-order Horner's rule, we observe a gain of 13 cycles, that is, a speedup of about 35 % compared to Horner's rule. Moreover, the evaluation of the polynomial P in (5.1) is done in 23 cycles, with an evaluation error no larger than η in (5.3). More precisely, we may check with Gappa that the evaluation error satisfies

$$\begin{aligned} \rho(\mathcal{P} &\leq 321377975215324845 \times 2^{-86} \\ &\lesssim 2^{-27.843}, \end{aligned}$$

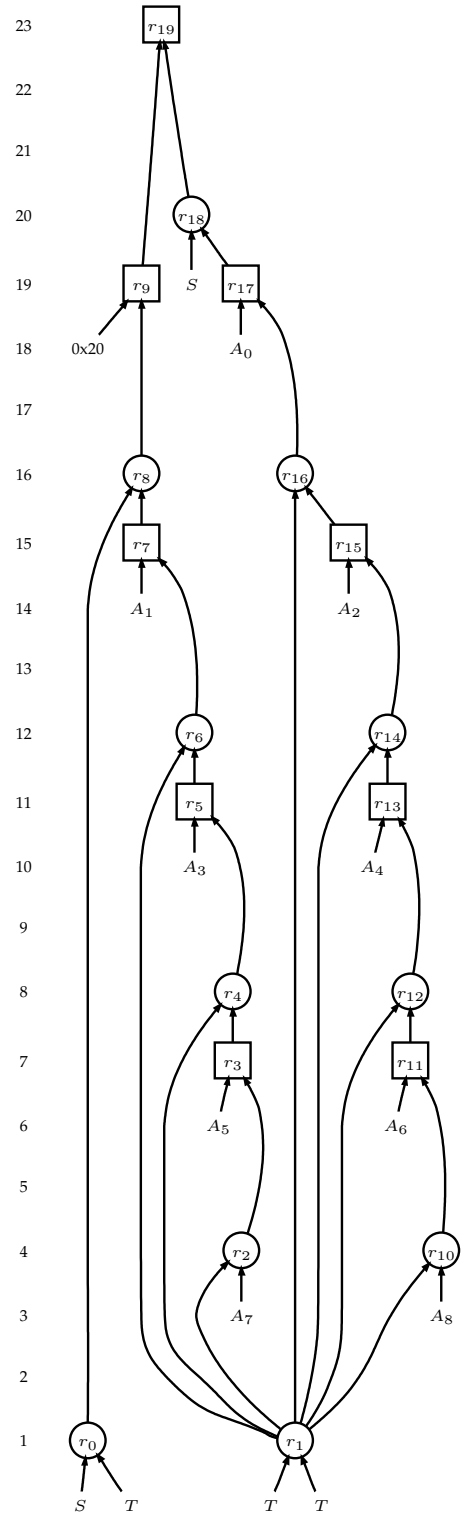


Figure 5.1: Evaluation tree for square root implementation using *second-order Horner's rule.*

and conclude that this evaluation program could have also been used for implementing *binary32* correctly-rounded square root.

5.1.3 Estrin's method

The last classical evaluation scheme we present here is Estrin's method. This evaluation scheme is based on the "divide and conquer" strategy and consists in splitting the polynomial to be evaluated in its high and low parts, evaluating them using Estrin's method, and combining them together using a last step involving one multiplication by a power of x, and one addition [Knu98, §4.6.4]. In fact, Estrin's method is called recursively on the high and low parts of the polynomial to be evaluated, until having degree-1 polynomials.

The main advantage of this evaluation approach relies on the fact that it is highly parallel, and it tends to expose lots of instruction-level parallelism. For example, let $p(x)$ be a degree-7 polynomial with coefficients p_i given in the monomial basis. Table 5.2 below shows a feasible scheduling for evaluating that polynomial on the ST231 processor.[2] However the speedup in

Cycle	Issue 1	Issue 2	Issue 3	Issue 4
0	$r_1 = p_7 \cdot x$		$r_2 = x \cdot x$	
1	$r_3 = p_5 \cdot x$		$r_4 = p_3 \cdot x$	
2	$r_5 = p_1 \cdot x$			
3	$r_6 = r_2 \cdot r_2$	$r_7 = r_1 + p_6$		
4	$r_8 = r_7 \cdot r_2$	$r_9 = r_3 + p_4$	$r_{10} = r_4 + p_2$	
5	$r_{11} = r_5 + p_0$	$r_{12} = r_{10} \cdot r_2$		
6				
7	$r_{13} = r_8 + r_9$			
8	$r_{14} = r_{13} \cdot r_6$	$r_{15} = r_{12} + r_{11}$		
9				
10				
11	$a(x) = r_{14} + r_{15}$			

Table 5.2: Degree-7 polynomial evaluation using Estrin's rule on ST231.

terms of evaluation latency on unbounded parallelism is possible only to the detriment of an increase of the total number of operations. Indeed, for $n+1$ a power of 2, it evaluates a degree-n polynomial in n additions and $n + \log(n+1) - 1$ multiplications [Rev06]. For example, for evaluating a degree-7 polynomial as described in Table 5.2 above, we need 7 additions and 9 multiplications, instead of 7 additions and 7 multiplications with Horner's rule.

Let us return to the bivariate polynomial $P(s,t) = 2^{-p-1} + s \cdot a(t)$ defined in (5.1). The first idea consists in evaluating the polynomial $a(t)$ using the Estrin's method, and then applying one step of Horner's rule, as shown below:

$$\begin{aligned} P(s,t) &= 2^{-25} + s \cdot \Big[\big((a_0 + t \cdot a_1) + t^2 \cdot (a_2 + t \cdot a_3)\big) + \\ &\qquad (t^2 \cdot t^2) \cdot \Big(\big((a_4 + t \cdot a_5) + t^2 \cdot (a_6 + t \cdot a_7)\big) + (t^2 \cdot t^2) \cdot a_8\Big)\Big]. \end{aligned} \tag{5.4}$$

An implementation of this evaluation scheme is shown in Listing 5.5 below.

Figure 5.2 displays the evaluation tree corresponding to (5.4). We can observe that the critical path of the evaluation is composed by the computation of $r_2 = t^2$, and then the computation of r_7

[2] On the ST231, additions and subtractions have a latency of 1 cycle, multiplications have a latency of 3 cycles, and 2 pipelined multiplications can be launched every cycle.

```
1  uint32_t __sqrt_eval__estrin__(uint32_t T, uint32_t S)
2  {
3    uint32_t r0  = mul(T, 0x3ffffafc);     // 1.31
4    uint32_t r1  = 0x80000007 + r0;        // 1.31
5    uint32_t r2  = mul(T, T);              // 0.32
6    uint32_t r3  = mul(T, 0x07f9a6be);     // 1.31
7    uint32_t r4  = 0x0fff6f59 - r3;        // 1.31
8    uint32_t r5  = mul(r2, r4);            // 1.31
9    uint32_t r6  = r1 - r5;                // 1.31
10   uint32_t r7  = mul(r2, r2);            // 0.32
11   uint32_t r8  = mul(T, 0x0304d2f4);     // 1.31
12   uint32_t r9  = 0x04db72ce - r8;        // 1.31
13   uint32_t r10 = mul(T, 0x0093fa25);     // 1.31
14   uint32_t r11 = 0x0198e4c7 - r10;       // 1.31
15   uint32_t r12 = mul(r2, r11);           // 1.31
16   uint32_t r13 = r9 + r12;               // 1.31
17   uint32_t r14 = mul(r7, 0x0019b4c0);    // 1.31
18   uint32_t r15 = r13 + r14;              // 1.31
19   uint32_t r16 = mul(r7, r15);           // 1.31
20   uint32_t r17 = r6 - r16;               // 1.31
21   uint32_t r18 = mul(S, r17);            // 2.30
22   uint32_t r19 = 0x00000020 + r18;       // 2.30
23   return r19;
24 }
```

Listing 5.5: Polynomial evaluation with Estrin's method in unsigned fixed-point arithmetic.

to r_{19}: it has a length of 18 cycles (assuming as before addition in 1 cycle and multiplication in 3 cycles). Using this parenthesization enables to use more than 2 issues on the ST231, especially at the beginning of the evaluation, since several parts of the polynomial can be evaluated in parallel. Remark that, in terms of accuracy, this parenthezisation entails an evaluation error no larger than the required bound η in (5.3), and more particulary:

$$\begin{aligned} \rho(\mathcal{P}) &\leq 1103468431366337355 \times 2^{-88} \\ &\lesssim 2^{-28.0633}. \end{aligned} \tag{5.5}$$

However, we observe in (5.4) and on Figure 5.2 that the last multiplication by s is a bottleneck in the evaluation of P, and we can distribute it over the evaluation of $a(t)$. This gives the following parenthesization, which we shall call "second Estrin's method":

$$\begin{aligned} P(s,t) = 2^{-25} + \Big[& s \cdot \big((a_0 + t \cdot a_1) + t^2 \cdot (a_2 + t \cdot a_3)\big) + \\ & \big(s \cdot (t^2 \cdot t^2)\big) \cdot \Big(\big((a_4 + t \cdot a_5) + t^2 \cdot (a_6 + t \cdot a_7)\big) + (t^2 \cdot t^2) \cdot a_8\Big)\Big]. \end{aligned} \tag{5.6}$$

This parenthesization leads to an implementation in 15 cycles, that is, 3 cycles less than the one with Estrin's method, and with an evaluation error $\rho(\mathcal{P})$ satifying:

$$\begin{aligned} \rho(\mathcal{P}) &\leq 1103468431252881531 \times 2^{-88} \\ &\lesssim 2^{-28.0633}, \end{aligned} \tag{5.7}$$

which is less than the required bound $\eta \approx 2^{-25.30}$ in (5.3). Figure 5.2 displays the evaluation tree corresponding to (5.6).

We conclude that Estrin's methods, as well as Horner's rules presented above in the section, could have been used for implementing the *binary32* correctly-rounded square root in FLIP. However, even if we use the *second Estrin's method*, which is the fastest of all these 4 evaluation

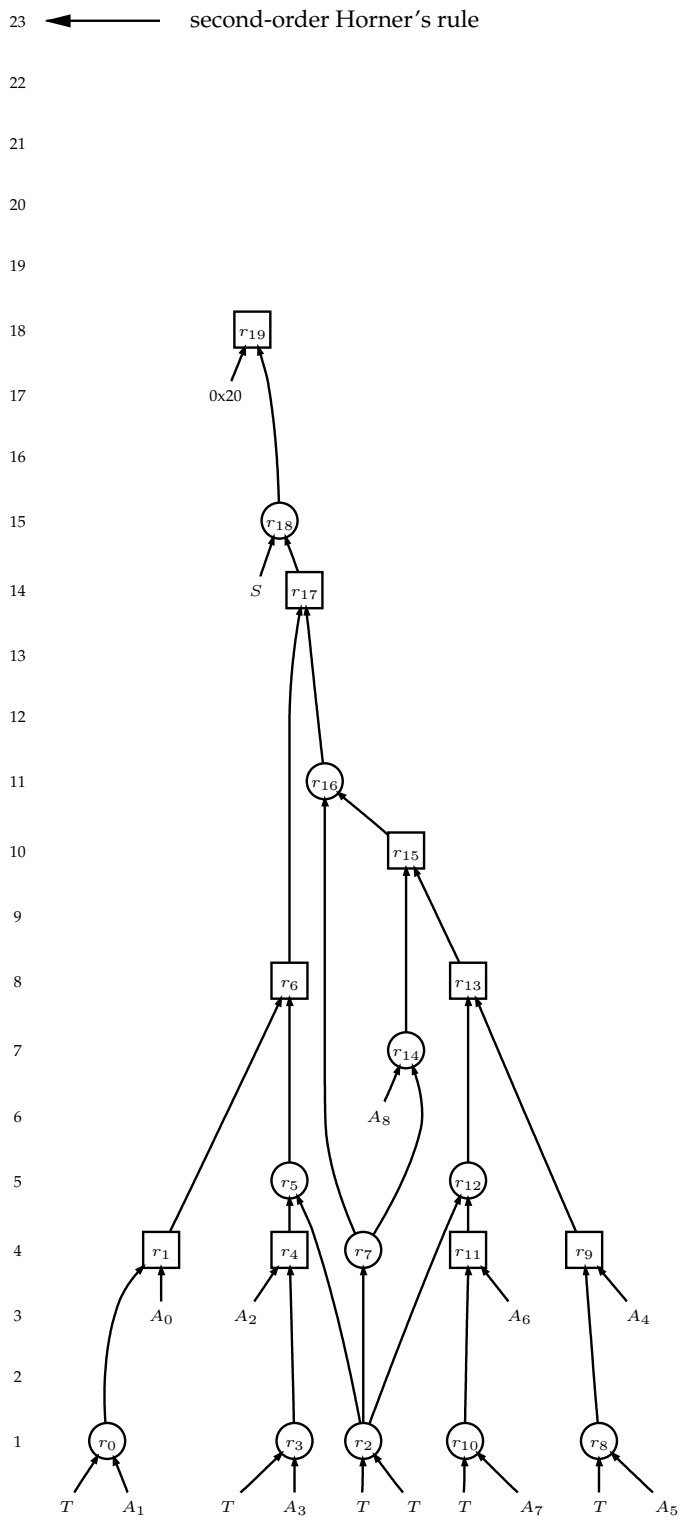


Figure 5.2: Evaluation tree for square root implementation using *Estrin's method*.

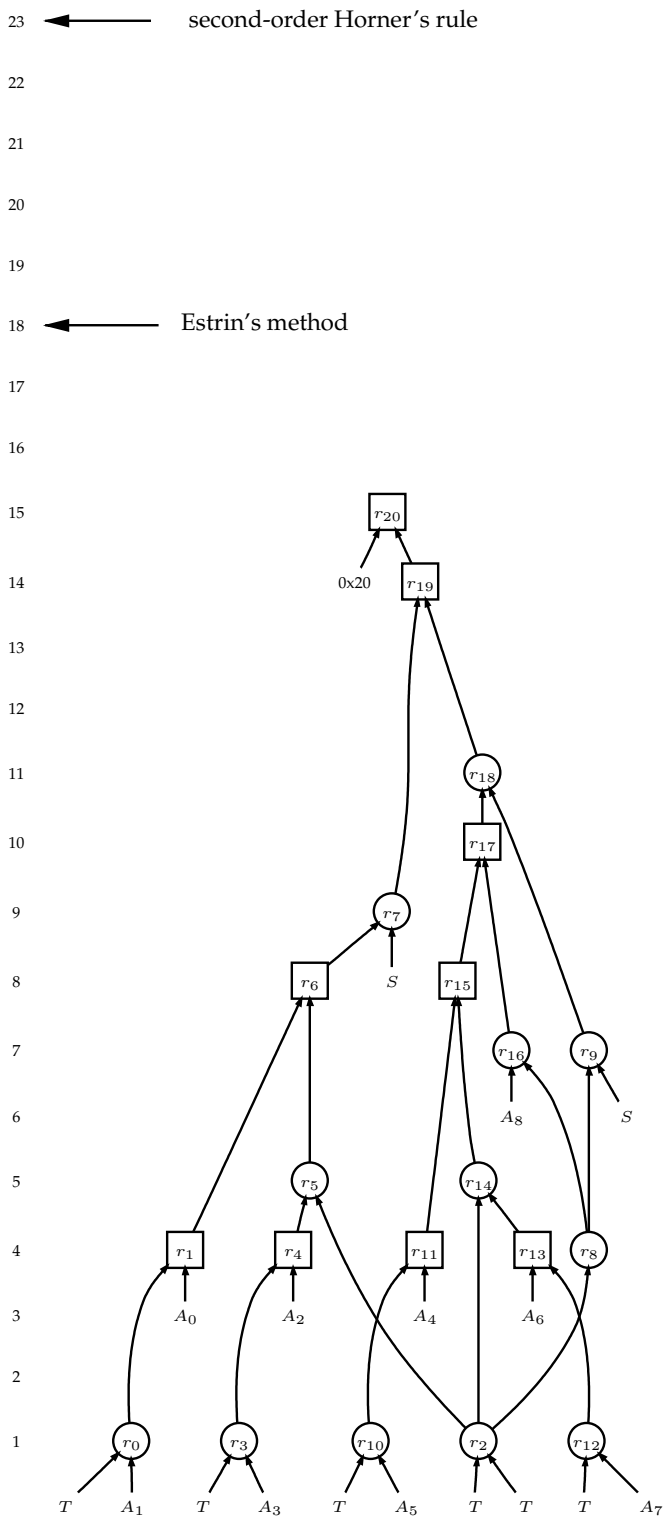


Figure 5.3: Evaluation tree for square root implementation using *second Estrin's method*.

schemes, we have an implementation in 15 cycles, that is, more than 15 % slower than the fastest one found with CGPE (which has a latency of 13 cycles and is displayed in Listing 3.3), and effectively used for implementing the square root function in FLIP.

Let us now have a look at the interest of using these classical evaluation methods for implementing functions other than square root.

5.1.4 Numerical results on fixed-point polynomial evaluation on ST231 processor

In the three previous subsections, we have detailed how to write efficient implementation using classical evaluation schemes, and given examples of code for the *binary32* implementation of the correctly-rounded square root function. We have concluded that these evaluation programs could have been used for implementing this function in FLIP. Let us now see if these evaluation schemes can also be used for implementing the others functions of FLIP. To do so, let us consider the function $x^{1/n}$ for $n \in \{-4, -3, -2, -1, 2, 3, 4\}$ as well as division.

We have generated with the script in Listing 3.1, Chapter 3, the polynomial approximants together with the certified evaluation bounds η sufficient to ensure correct-rounding for such functions, and used the approach presented in this chapter (in the paragraphs "Unsigned fixed-point evaluation and arithmetic operator choice" and "Evaluation program validation") to write the evaluation codes. Therefore, Table 5.3 below displays, for each function, the degree of the polynomial approximant $a(t)$ in (5.1), as well as the bound η. And for each evaluation scheme, it gives the evaluation error bound computed with Gappa. In Table 5.3, "-" means that the arithmetic operator choice cannot be done automatically using naive interval arithmetic (that is, we cannot ensure that the evaluation can be done in unsigned fixed-point arithmetic), and the error bounds that are underlined are those which are not sufficient to ensure correct rounding.

	$x^{1/2}$	$x^{-1/2}$	$x^{1/3}$	$x^{-1/3}$	$x^{1/4}$	$x^{-1/4}$	x^{-1}	x/y
Degree	8	9	8	9	8	8	10	10
Required approx. error	$2^{-25.51}$	$2^{-25.52}$	$2^{-25.68}$	$2^{-25.68}$	$2^{-25.76}$	$2^{-25.77}$	2^{-25}	$2^{-26.99}$
Bound on θ	$2^{-27.99}$	$2^{-26.60}$	$2^{-27.76}$	$2^{-27.76}$	$2^{-27.74}$	$2^{-25.95}$	$2^{-26.39}$	$2^{-27.41}$
Bound on η	$2^{-25.30}$	$2^{-25.94}$	$2^{-25.40}$	$2^{-25.41}$	$2^{-25.43}$	$2^{-28.09}$	$2^{-25.69}$	$2^{-26.99}$
Horner's rule	$2^{-28.06}$	$2^{-28.06}$	$2^{-27.93}$	$2^{-27.93}$	$2^{-27.87}$	$\underline{2^{-27.87}}$	$2^{-27.67}$	$2^{-27.41}$
2nd-order Horner's rule	$2^{-27.84}$	-	$2^{-27.78}$	-	$2^{-27.75}$	-	-	-
Estrin's rule	$2^{-28.06}$	$2^{-28.65}$	$2^{-27.93}$	$2^{-27.76}$	$2^{-27.87}$	$\underline{2^{-27.77}}$	$2^{-27.03}$	$\underline{2^{-26.57}}$
2nd Estrin's rule	$2^{-28.06}$	$2^{-27.55}$	$2^{-27.93}$	$2^{-27.65}$	$2^{-27.87}$	$\underline{2^{-27.70}}$	$2^{-27.03}$	$\underline{2^{-26.48}}$

Table 5.3: Evaluation error bounds for various functions and various fixed-point evaluation schemes.

We observe from Table 5.3 above, that the classical methods presented in this chapter can be effectively used for implementing the first nth roots for $n \in \{2, 3, 4\}$. Also, some of them, but not all, can be used for implementing their reciprocals, and inversion and division. More precisely, from these results, we can make the following remarks:

- Horner's rule can be used for the implementation of all these functions, but the reciprocal fourth root. For $n = -4$, the polynomial $a(t)$ approximates the function $2^{1/4} \cdot (1 + x)^{-1/4}$ with an approximation error less than $\theta \approx 2^{-25.95}$, which is slightly smaller than required bound $\approx 2^{-25.77}$. In this case, the polynomial has to be evaluated very accurately, with an

evaluation error $\lesssim 2^{-28.09}$. Unfortunately this bound cannot be satisfied using one of the classical methods presented in this chapter. It turns out that for implementing the reciprocal fourth root, we should take a polynomial approximant $a(t)$ of higher degree, or find another evaluation program, using CGPE for example, whose evaluation error satisfies the bound.

- In our context, we cannot write code automatically to evaluate polynomials whose coefficients alternate completely, and more particularly we cannot ensure using interval arithmetic that all intermediate variables are always + or always -. To understand this, let us consider the case where $|n| = 2$. Table 5.4 below shows the sign changing of the polynomial coefficients for $n = 2$ and $n = -2$. (Recall that $a(t)$ has degree 8 when $n = 2$ and degree 9 when $n = -2$.) When $n = -2$, "second-order" Horner's rule will evaluate the even and odd parts separately, whose results, respectively, are positive and negative. Hence, using interval arithmetic will lead to a result, whose sign changes when the input varies. On the contrary, when $n = 2$, since $a_0 < a_1 < \cdots < a_8$, we may check that the even and odd parts are both positive, and, in interval arithmetic, the result does not change of sign when input varies. Actually, we can observe this phenomenum for all the cases where "second-order" Horner's rule cannot be used in Table 5.3.

	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9
$n = 2$	+	+	-	+	-	+	-	+	-	
$n = -2$	+	-	+	-	+	-	+	-	+	-

Table 5.4: Signs of the coefficients of polynomial approximant $a(t)$, when $|n| = 2$.

- Remark also that our division algorithm of Chapter 4 could have been implemented using Horner's rule. But on the ST231 the evaluation would have been in 44 cycles, instead of 14 cycles for the current implementation of division, that is, more than more 3 times slower. Remark also that Estrin-based rules could not have been used directly for implementing our *binary32* division, since the evaluation error cannot be checked with Gappa.

5.2 Polynomial evaluation via coefficient adaptation

Until now we have presented some classical methods for evaluating a degree-n polynomial and how they can be used for evaluating polynomials (univariate or bivariate) on integer architectures, and especially on the ST231, through various examples. These methods are more or less efficient, in terms of evaluation latency and accuracy, and we have also explained why they can or cannot be used in our context for implementing mathematical functions. The efficiency of these schemes in terms of latency is always due to a higher instruction-level parallelism exposure.

However, recall that in the 60's, multiplication was much slower than addition. This statement has motivated the implementation of evaluation schemes that reduce the number of multiplications, to the detriment of possible extra (but much cheaper) additions. Hence, depending on the context, these new evaluation schemes can be expected to be faster to be evaluated. Since multiplication is slower on the ST231 processor than addition, what about using these methods on this architecture?

This section presents some examples of polynomial evaluation schemes based on the *adaptation* of the polynomial coefficients to be evaluated, and explains why we have not used them for implementing mathematical functions.

5.2.1 What is coefficient adaptation?

From the coefficients of the polynomial in the monomial basis, *coefficient adaptation* consists in computing new coefficients, a new expression for the polynomial that involves fewer operations (in particular fewer multiplications). This phase may be costly, but is done once for all the evaluations. (See [Knu62], [Fik67], or [Knu98] for examples of adaptations.)

The first method based on coefficient adaptation is due to Motzkin [Knu98, p. 490]. It enables to evaluate a degree-4 polynomial $a(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2 + a_3 \cdot x^3 + a_4 \cdot x^4$ as follows:

$$a(x) = \big((y + x + \alpha_2) \cdot y + \alpha_3\big) \cdot \alpha_4 \quad \text{with} \quad y = (x + \alpha_0) \cdot x + \alpha_1, \tag{5.8}$$

with

$$\begin{cases} \alpha_0 = \frac{1}{2} \cdot (a_3/a_4 - 1), & \beta = a_2/a_4 - \alpha_0 \cdot (\alpha_0 + 1), \\ \alpha_1 = a_1/a_4 - \alpha_0 \cdot \beta, & \alpha_2 = \beta - 2 \cdot \alpha_1, \\ \alpha_3 = a_0/a_4 - \alpha_1 \cdot (\alpha_1 + \alpha_2), & \alpha_4 = a_4. \end{cases}$$

Once we have computed the $\alpha_i's$, the evaluation of the polynomial $a(x)$ in (5.8) can be done using 5 additions, but only 3 multiplications, instead of 4 additions and 4 multiplications with Horner's rule. On unbounded parallelism, the critical path consists in computing y in 2 additions and 1 multiplication, and $a(x)$ in 2 additions and 2 multiplications (since $x + \alpha_2$ may be computed in parallel of y), that is, the critical path has a length of 4 additions and 3 multiplications. This is less than than the critical path of Horner's rule, which consists of 4 additions and 4 multiplications.

5.2.2 Knuth and Eve's algorithm

The first algorithm based on coefficient adaptations presented in this section is Knuth and Eve's algorithm. Introduced in the 60's by Knuth [Knu98, §4.6.4, Theorem E], it has been completed by Eve [Eve64] in 1964 (see also [Mul06, §3.8] or [Rev06, §2] for details). When $n \geq 3$, it enables to evaluates a univariate degree-n polynomial in at most n additions and $\lfloor n/2 \rfloor + 2$ multiplications, that is, roughly twice fewer multiplications than Horner's rule.

Coefficient adaptation using Knuth and Eve's algorithm

Let $a(x)$ be a univariate degree-n polynomial to be adapted, $n \geq 3$, and let $p(x)$ be a univariate degree-n polynomial defined as $p(x) = a(x - c)$, with $c \in \mathbb{R}$. Before seing the interest of the value c and explain how to choose it, let us first show how to adapt the polynomial $a(x)$, and more particularly, the corresponding polynomial $p(x)$. The idea of this algorithm consists in splitting the polynomial $p(x)$ into its even and odd parts, g and h, respectively, so that:

$$p(x) = g(y) + x \cdot h(y), \quad \text{with} \quad y = x^2. \tag{5.9}$$

Assume that $h(y)$ has m real roots α_i's, that is,

$$h(x) = p_{2m+1} \cdot (y - \alpha_{m-1}) \cdots (y - \alpha_0), \quad \text{with} \quad m = \lfloor (n-1)/2 \rfloor,$$

with p_{2m+1} the leading coefficient of $p(x)$. The evaluation of $p(x)$ using Knuth and Eve's algorithm is done as follows:

$$p(x) \quad = \quad \Big(\dots \big((g^{(m)}(y) + p_{2m+1} \cdot x)(y - \alpha_{m-1}) + \beta_{m-1} \big) \cdots \Big)(y - \alpha_0) + \beta_0,$$

with the coefficients β_i's defined as the successive remainders of the division of $g^{(i)}(y)$ by $y - \alpha_i$, where

$$g^{(i)}(y) = g^{(i+1)}(y) \cdot (y - \alpha_i) + \beta_i \quad \text{and} \quad g^{(0)}(y) = g(y),$$

and

$$g^{(m)}(y) = \begin{cases} g_0^{(m)}, & \text{if } n \text{ is odd,} \\ g_1^{(m)} y + g_0^{(m)}, & \text{if } n \text{ is even,} \end{cases} \tag{5.10}$$

with $g_1^{(m)}$ and $g_0^{(m)}$ being rational coefficients. Finally, evaluating $a(x)$ using this approach is done by evaluating $p(x+c)$.

Let us now detail the role played by the value c. In fact this *shift* is here chosen so that the polynomial $h(y)$ in (5.9) has exactly m real roots. In [Eve64], Eve has shown that if a degree-n polynomial has at least $n-1$ roots whose real parts are all nonnegative or nonpositive, then its odd part has only real roots. Hence in this case, we have to choose c so that $p(x)$ satisfies this condition. Knuth proposed in [Knu98, §4.6.4, Theorem E] his own value c, so that the coefficient β_0 equals 0 (which saves one addition).

Remark here that this adaptation is not rational, since in general the computation of the real roots α_i's cannot be done exactly. Combining this remark with the fact that several values c may be chosen, and that the α_i's may be handled in $m!$ ways, we conclude that many different adaptations with different numerical accuracies may be chosen (see [Rev06] for examples).

Why we have *not* chosen Knuth and Eve's algorithm for FLIP?

Recall that the polynomial $P(s,t)$ defined in Chapter 3 for the implementation of the square root is defined as follows

$$P(s,t) = 2^{-25} + s \cdot a(t),$$

with $a(t)$ a degree-8 univariate polynomial. The evaluation of this bivariate polynomial P using Knuth and Eve's algorithm can be done by adapting the polynomial $a(t)$ and then by applying one step of Horner's rule, as follows:

$$P_{\text{KE}}(s,t) = 2^{-25} + s \cdot \Big[\Big(\big(\big((g^{(3)}(y) + p_7 \cdot (x+c))(y-\alpha_2) + \beta_2\big)(y-\alpha_1) + \beta_1\big)\Big)(y-\alpha_0) + \beta_0\Big],$$

with $g^{(3)}(y) = g_1^{(3)} y + g_0^{(3)}$, $y = (x+c)^2$ and $c \in \mathbb{R}$, and the α_i's and the β_i's being the new coefficients computed as presented above.

Let us now see the length of the critical path on unbounded parallelism. It consists of:

- the computation of $y = (x+c)^2$ (1 addition, 1 multiplication),
- the evaluation of $g^{(3)}$ (1 addition, 1 multiplication),
- the addition to $p_7 \cdot (x+c)$ (1 addition),
- and 4 steps of 1 multiplication and 1 addition.

That is, assuming an addition in 1 cycle and a multiplication in 3 cycles, the critical path is of 25 cycles, which is almost twice the latency of the best evaluation program we have found for evaluating the polynomial P in FLIP.

Concerning the numerical accuracy of this scheme, we have adapted the polynomial $a(t)$ useful for the implementation of the square root in Chapter 3 with Sollya [Che09], [Lau08], [CL], in larger precision. With a shift $c \approx -2.5$, as prescribed by Eve, and for all the $3! = 6$ permutations of α_i's, the certified approximation error computing using supremum norm algorithm is of the order of $2^{-19.57}$, which is not enough compared to the required bound $(2^{-25} - 2^{-31.5})/2^{1/2} \approx 2^{-25.51}$.

5.2.3 Paterson and Stockmeyer's algorithm

This section presents another algorithm based on coefficient adaptation, due to Paterson and Stockmeyer [PS73] (see also [Rev06, §3]). It enables to evaluate a degree-n polynomial in at most $3n/2$ additions and $n/2 + \log_2(n)$ multiplications, that is, it indeed reduces the number of multiplications, but to the detriment of an increase of the number of additions. Let us now see how to adapt a given polynomial for this algorithm.

Coefficients adaptation using Paterson and Stockmeyer's algorithm

To be adapted with Paterson and Stockmeyer's method, the polynomial has to be *monic*, that is, the leading coefficient has to be equal to 1. If this is not the case, we first compute the corresponding monic polynomial (simply by dividing by the leading coefficient), and an extra multiplication will be needed during evaluation in order to get back to the initial polynomial. This algorithm is also based on the "divide and conquer" strategy, like Estrin's method. However, at a given level, when we will split up the polynomial to be evaluated, we will adapt its low part so that both subpolynomials (high and low parts) remain monic. Moreover, in the same way as Estrin's method, if the polynomial to be adapted is of degree $n \neq 2^p - 1$, then it is split up into polynomials of degree $2^i - 1$ according to the binary expansion of n, which will adapted using Paterson and Stockmeyer's algorithm. Then $\log_2(n)$ extra multiplications will be needed to determine the final result [PS73]. Consequently, in the sequel we assume with no loss of generality that $n + 1$ is a power of 2.

Let a be a univariate degree-n polynomial, with n of the form $n = 2^p - 1$ ($p \in \mathbb{N} \setminus \{0\}$). Paterson and Stockmeyer's algorithm evaluates a as follows

$$a(x) = (x^m + \alpha) \cdot \left(x^{m-1} + \sum_{i=0}^{m-2} a_{i+m} x^i \right) + \left(x^{m-1} + \sum_{i=0}^{m-2} \beta_i x^i \right), \quad \text{with} \quad m = (n+1)/2, \quad (5.11)$$

and $\alpha = a_{m-1} - 1$ and $\beta_i = a_i - \alpha \cdot a_{m-i}$. Then, each subpolynomial is evaluated recursively using the same splitting.

Why we have *not* chosen Paterson and Stockmeyer's algorithm for FLIP?

Again, let us consider the polynomial $P(s,t)$ defined in Chapter 3 for the implementation of the square root: $P(s,t) = 2^{-p-1} + s \cdot a(t)$, with $a(t)$ a univariate degree-8 polynomial. The evaluation of P via Paterson and Stockmeyer's method gives the following parenthesization:

$$\begin{aligned} P_{\text{PS}}(s,t) = 2^{-25} \quad &+ \quad s \cdot \Big[a_7 \cdot \Big((x^4 + \alpha_0) \cdot \big((x^2 + \alpha_1) \cdot (x - \beta_0) + (x + \beta_1) \big) \\ &+ \quad \big((x^2 + \alpha_2) \cdot (x + \beta_2) + (x - \beta_3) \big) \Big) + (a_8 \cdot x^8) \Big], \end{aligned}$$

Assuming unbounded parallelism, it follows that the critical path is given by

- the computation of $x^2 + \alpha_1$ (1 addition, 1 multiplication),
- the multiplication by $x + \beta_0$,
- the addition to $x + \beta_1$,
- 1 multiplication and 1 addition to compute the inner part of the parenthezisation,
- the multiplication by a_7 and the addition of $a_8 \cdot x^8$,

Coefficient	Sign	Value	Encoding integer	Format
a_8	-	0.00078448676504194736480712890625	`0x00336981`	$Q_{0.32}$
a_7	+	0.00451590376906096935272216796875	`0x0127f44b`	$Q_{0.32}$
α_0	+	1.27970726117491722106933359375	`0xccc0cf36`	$Q_{4.28}$
α_1	+	4.22259075008332729339599609375	`0x871f76a3`	$Q_{3.29}$
β_0	-	2.76322183944284915924072265625	`0xb0d8a06a`	$Q_{2.30}$
β_1	+	3.26497823186218738555908203125	`0xd0f56742`	$Q_{2.30}$
α_2	+	4.28857989907264709472656250	`0xab8b0ee4`	$Q_{6.26}$
β_2	+	7.68501790426671504974365234375	`0xf5ebaaab`	$Q_{3.29}$
β_3	-	0.60501976870000362396240234375	`0x9ae29358`	$Q_{0.32}$

Table 5.5: New coefficients for evaluating $P(s,t)$ using Paterson and Stockmeyer's algorithm, for our *binary32* square root implementation.

- and finally one step of Horner's rule (1 multiplication and 1 addition).

Hence, assuming an addition in 1 cycle and a multiplication in 3 cycles, the length of the critical path of the evaluation of $P_{\text{PS}}(s,t)$ is 20 cycles, that is 7 cycles more than the best evaluation programs found with CGPE. We conclude first that in terms of evaluation latency, Paterson and Stockmeyer's method is not the most efficient one in the context of the ST231 processor.

Finally, let us see if Paterson and Stockmeyer's algorithm leads to a polynomial which, after adaptation, is accurate enough, that is, with an approximation error less that the required bound $(2^{-25} - 2^{-31.5})/2^{1/2} \approx 2^{-25.51}$. This adaptation, on the contrary to the Knuth and Eve's algorithm, is rational, and have been implemented exactly using Sollya. Then the absolute value of each coefficient has been truncated after 32 bits. We obtained a new polynomial that approximates the function $(1+x)^{1/2}$ with an approximation error $\approx 2^{-24.15}$. That means that it is also not sufficient to ensure correct rounding for square root. Moreover, Table 5.5 above displays the new coefficients computed for evaluating $P(s,t)$ using Paterson and Stockmeyer's algorithm. We remark that the coefficients α_i's and β_i's have different orders of magnitude. And as we have seen in Chapter 1, the computation in fixed-point arithmetic with numbers of different orders of magnitude leads to a much higher loss of accuracy.

Conclusion

We have presented in the section two evaluation methods that enable to evaluate a polynomial by preliminarily adapting its coefficients. The objective was to reduce the number of multiplications (sometimes at the expense of the number of additions). However, we have seen on the example of the implementation of the square root, that both methods are ill-suited in our context. First, even if they involve fewer multiplications, the critical path of the evaluation is longer than the one of the evaluation program chosen for the square root in Chapter 3. Secondly, once the polynomial is adapted, its approximation error is larger than the error required for ensuring correct rounding.

This is why, to implement the mathematical functions of FLIP (based on polynomial evaluation), we have chosen to use evaluation methods *without* adaptation of coefficients.

CHAPTER 6

Computing efficient polynomial evaluation programs

This chapter presents the methodology that we have implemented for generating efficient *and* certified *evaluation programs. More particularly, it is based on algorithms that compute all the evaluation schemes (parenthesizations) for evaluating a given bivariate polynomial, the determination of a lower bound of the minimal evaluation latency, and a heuristic for finding* best *parenthesizations of the polynomial. Since the number of such schemes may be extremely large (at least for degrees* ≥ 5*), these heuristics enable to converge quickly towards* efficient *schemes (on unbounded parallelism). This methodology have been integrated into the software environment CGPE (Code Generation for Polynomial Evaluation), which has already been used for generating the evaluation schemes used by several functions of the FLIP library, like roots and division (Chapters 3 and 4, respectively). For some of them, we can conclude that the program generated is optimal in term of evaluation latency.*

This chapter presents the algorithms we have implemented to generate all the evaluation schemes, or converge toward the most *efficient* ones, for the evaluation of univariate and bivariate polynomials, using only additions and multiplications, and without any preliminary *adaptation* of coefficients. Such evaluation schemes thus correspond to various parenthesizations of the arithmetic expression represented by a given polynomial. Also, by *efficient* we mean evaluation schemes that reduce the latency when assuming unbounded parallelism, that is, the length of the critical path of the evaluation. In this chapter, we focus on the generation of evaluation schemes, and we do not discuss their accuracy. For example, given a degree-n univariate polynomial $a(x)$ defined as follows

$$a(x) = a_0 + a_1 \cdot x + \cdots + a_n \cdot x^n \quad \text{with} \quad a_n \neq 0,$$

what are all the possible schemes to evaluate $a(x)$, by using only additions and multiplications, and without any coefficient adaptations? In this first example, we assume that $a_0 \neq 0$. This implies that $a(0) \neq 0$ and, consequently, we know that the last operation of the evaluation must be an addition of, let us say, two subexpressions $a'(x)$ and $a''(x)$, as shown in Figure 6.1. Hence in this case, it turns out that the goal consists in determining all the possible schemes to evaluate $a'(x)$ and $a''(x)$ such as

$$a(x) = a'(x) + a''(x).$$

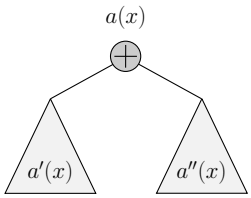


Figure 6.1: Evaluation tree of polynomial $a(x)$.

Horner's rule, presented in Section 5.1.1, gives in particular

$$a'(x) = a_0 \quad \text{and} \quad a''(x) = x \cdot \Big(a_1 + x \cdot \big(a_2 + \cdots + x \cdot (a_{n-1} + x \cdot a_n) \ldots\big)\Big).$$

At this time we consider unbounded parallelism and thus we know also from Chapter 5 that the above splitting given by Horner's rule does not give the lowest evaluation latency for $a(x)$. If we want to reduce the evaluation latency of $a(x)$, we have to choose another splitting that reduces the maximum of the evaluation latencies of $a'(x)$ and $a''(x)$. But how to determine such a *best* splitting?

Before presenting the algorithms, recall that in Part I, we have shown that the implementation of a mathematical function may be done via the evaluation of a particular bivariate polynomial P of the form $P(s,t) = 2^{-p-1} + s \cdot a(t)$, with $a(t)$ a univariate polynomial of smallest degree n:

$$P(s,t) = 2^{-p-1} + s \cdot (a_0 + a_1 \cdot t + \cdots + a_n \cdot t^n).$$

Since P has to be evaluated at runtime, the evaluation program has to be as fast as possible. To do that, we may first evaluate the polynomial $a(t)$ and then end by a last step (multiplication by s and addition of 2^{-p-1}). In [HKST99], a methodology is proposed for building optimal evaluation schemes for the evaluation of univariate polynomials on the Itanium® processor using only `fma` operations (see also [CHT02] for numerical results). This methodology has inspired the *brute force searching* presented in [Gre02] for generating polynomial evaluation parenthesizations using at best SIMD instructions, for the implementation of faster mathematical functions for the PlayStation® 2 (Sony). Of course, we do not have efficient `fma` operators on the ST231, but following this method, we may derive an approach adapted to our context. Moreover we know that distributing the multiplication by s over the evaluation of the polynomial $a(t)$ may lead to lower evaluation latencies (see Section 5.1 for examples). The problem is now to find all the possible evaluation schemes to evaluate polynomials of the form $P(s,t) = 2^{-p-1} + s \cdot a(t)$ (and thus not only univariate polynomials). Since the problem of evaluating this kind of polynomials is clearly a special case of the one of evaluating general bivariate polynomials, we have in fact generalized our algorithm to any bivariate polynomial.

This chapter is organized as follows. Section 6.1 describes the framework used for implementing the algorithms of evaluation scheme generation, presents and details the implementation of the building rules used for constructing such evaluation schemes, and then gives some numerical results. Section 6.2 explains how to compute parenthezisations of low latency only, by determining first a target latency. Section 6.3 presents a heuristic that enables to find quickly an efficient splitting of the polynomial, and then to converge toward an efficient evaluation scheme. Finally, Section 6.4 presents briefly the framework we have implemented to generate efficient and certified C codes for the implementation of mathematical functions. Parts of this framework have already been integrated into CGPE (Code Generation for Polynomial Evaluation).

6.1 Computing all the parenthesizations

This section defines the framework used for the generation of the evaluation schemes, and details the building rules used for generating all the parenthesizations. Finally, it gives an example of execution of the algorithm for the generation of all the 51 possible parenthesizations of general degree-2 bivariate polynomials.

6.1.1 Preliminary definitions

Let $a(x, y)$ be a bivariate degree-n polynomial defined as follows:

$$a(x,y) = \sum_{i=0}^{n_x} \sum_{j=0}^{n_y} a_{i,j} \cdot x^i \cdot y^j \qquad \text{with} \quad n = n_x + n_y, \quad \text{and} \quad a_{n_x,n_y} \neq 0, \tag{6.1}$$

with n the total degree of the polynomial. Recall that the goal is to build all the possible division free evaluation schemes that evaluate the polynomial $a(x, y)$, by using only additions, multiplications, and without any preliminary adaptation of the coefficients. More precisely, we want to build all the expressions, where only the coefficients $a_{i,j}$ of $a(x, y)$ in the monomials basis, the indeterminates x and y, and the operators addition of $(+)$ and multiplication ($\times$ or $\cdot$) appear, so that these expressions represent the polynomial $a(x, y)$ in (6.1). Here, addition and multiplication are commutative and associative; also multiplication is distributive over addition. Such expressions are also called *parenthesizations*. From now, we do not take into account neither the cost of the operators, nor the nature of the coefficients $a_{i,j}$ (integers, fixed-point numbers, or floating-point numbers).

Example 6.1. *Let $a(x)$ be a degree-2 univariate polynomial defined as*

$$a(x) = a_{0,0} + a_{1,0} \cdot x + a_{2,0} \cdot x^2.$$

We can check that $a(x)$ admits the following seven parenthesizations:

$$a(x) = \begin{cases} \big(a_{0,0} + (a_{1,0} \cdot x)\big) + \big((a_{2,0} \cdot x) \cdot x\big) \\ \big(a_{0,0} + (a_{1,0} \cdot x)\big) + \big(a_{2,0} \cdot (x \cdot x)\big) \\ a_{0,0} + \big(x \cdot (a_{1,0} + a_{2,0} \cdot x)\big) \\ a_{0,0} + \big((a_{1,0} \cdot x) + \big((a_{2,0} \cdot x) \cdot x\big)\big) \\ a_{0,0} + \big((a_{1,0} \cdot x) + \big(a_{2,0} \cdot (x \cdot x)\big)\big) \\ (a_{1,0} \cdot x) + \big(a_{0,0} + \big((a_{2,0} \cdot x) \cdot x\big)\big) \\ (a_{1,0} \cdot x) + \big(a_{0,0} + \big(a_{2,0} \cdot (x \cdot x)\big)\big) \end{cases}$$

Remark that all these parenthesizations have been found using the algorithm presented in the remainder of this section.

For building such expressions, that is, such parenthesizations, we have implemented an iterative process that builds, at each step i ($i \geq 1$), all the expressions of total degree i that can be used for evaluating the polynomial $a(x, y)$ in (6.1). Such expressions are called *valid*. Definition 6.1 below defines precisely what we call a valid expression.

Definition 6.1. *(valid expression) Let e be an arithmetic expression. We consider that e is* valid *if and only if it exists a subexpression p of $a(x, y)$, such that*

$$a(x,y) = e \cdot x^i y^j + p, \quad \textit{with} \quad i, j \geq 0. \tag{6.2}$$

Hence, it follows that at step n (with n the total degree of the polynomial to be evaluated), we will have generated all the parenthesizations for evaluating $a(x, y)$.

Example 6.2. *Let $a(x)$ be the degree-2 univariate polynomial defined in Example 6.1. At step 1, the algorithm builds all the expressions of degree 1 that can be used for evaluating $a(x)$, that is:*

$$a_{1,0} \cdot x, \quad a_{2,0} \cdot x, \quad a_{0,0} + (a_{1,0} \cdot x), \quad \text{and} \quad a_{1,0} + (a_{2,0} \cdot x),$$

so that all the expressions in Example 6.1 can be generated at step 2.

To do so, let us first define two sets.

- $\mathrm{E}^{(k)}$ is the set of *valid* expressions of total degree exactly k, that is, that enable to evaluate the polynomial $a(x, y)$ and where at least one coefficient $a_{i,j}$ of the polynomial appears;
- $\mathrm{P}^{(k)}$ is the set of expressions of the form $x^i y^j$, with $i + j = k$, such that $0 \leq i \leq n_x$ and $0 \leq j \leq n_y$, where n_x and n_y are as in (6.1).

For a given k, all the expressions of $\mathrm{E}^{(k)}$ and $\mathrm{P}^{(k)}$ differ from each other. It follows that $\mathrm{E}^{(0)}$ contains all the cofficients $a_{i,j}$ of the polynomial, and that $\mathrm{P}^{(1)}$ contains the indeterminates x and y. In the Example 6.2, we have:

$$\mathrm{E}^{(0)} = \{a_{0,0}, a_{1,0}, a_{2,0}\}, \quad \mathrm{P}^{(1)} = \{x\}, \quad \text{and} \quad \mathrm{E}^{(1)} = \{a_{1,0} \cdot x, a_{2,0} \cdot x, a_{0,0} + (a_{1,0} \cdot x), a_{1,0} + (a_{2,0} \cdot x)\}.$$

In the remainder of this chapter, we will denote by *expressions* the elements of $\mathrm{E}^{(k)}$ and by *powers* the elements of $\mathrm{P}^{(k)}$. It remains now to explain how to build, at step k, all the expressions of $\mathrm{E}^{(k)}$ and powers of $\mathrm{P}^{(k+1)}$ by combining the expressions of $\mathrm{E}^{(j)}$ and $\mathrm{P}^{(j+1)}$, with $j < k$.

6.1.2 Building rules

This section introduces the building rules used for generating all the expressions and the powers of degree k (that is, the expressions of $\mathrm{E}^{(k)}$ and the elements of $\mathrm{P}^{(k)}$). To do that, we assume that we have already computed the expressions of $\mathrm{E}^{(j)}$ and $\mathrm{P}^{(j)}$, for all $0 \leq j < k$.

Using (6.2), it follows that a power is obtained by multiplying two powers (Figure 6.2(a)), while a valid expression e can be built by multiplying an expression with a power (Figure 6.2(b)), or by adding two expressions (Figure 6.2(c)). The remainder of this section details the rule used for building powers, and the rules used for building expressions.

Rule R1 for building the powers. Let p be a power of the form $x^i y^i$, with $k = i + j$, that belongs to $\mathrm{P}^{(k)}$. It is obtained by multiplying two powers p_1 and p_2 of, respectively, $\mathrm{P}^{(k')}$ and $\mathrm{P}^{(k-k')}$ with $k' \leq \lfloor k/2 \rfloor$, of the form $p_1 = x^{i_x} y^{i_y}$ and $p_2 = x^{j_x} y^{j_y}$, such as

$$i = i_x + j_x \quad \text{and} \quad j = i_y + j_y,$$

and, when $k' = k/2$,

$$p_1 \cdot p_2 \in \mathrm{P}^{(k)} \quad \text{if and only if} \quad p_2 \cdot p_1 \notin \mathrm{P}^{(k)}. \tag{6.3}$$

Since multiplication is commutative, we can restrict to $k' \leq \lfloor k/2 \rfloor$, and the third condition in (6.3) enables no to generate doubles of powers when $k' = k/2$. Of course, this case does not occur when k is odd.

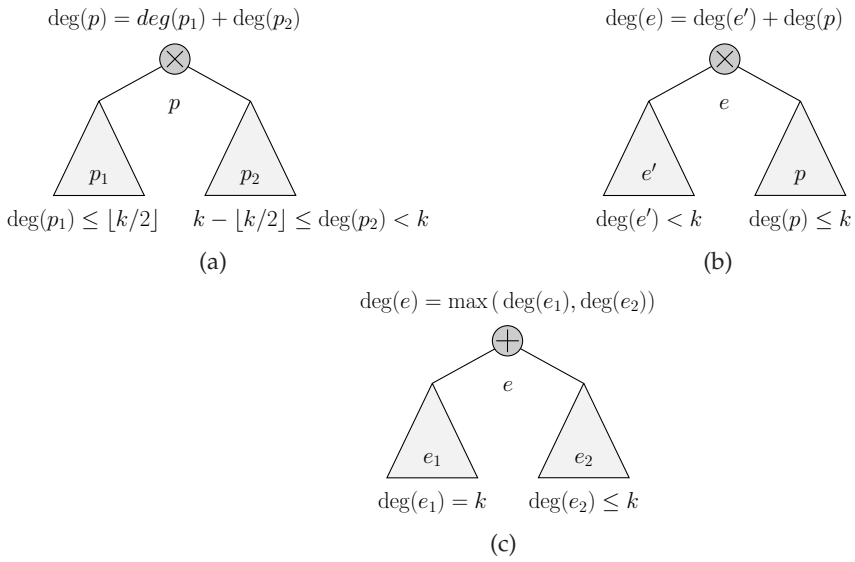


Figure 6.2: Building of a valid expression e or a power of degree k.

Rule R2 for building the expressions by multiplication. Let e be an expression of total degree k, that belongs to $\mathrm{E}^{(k)}$. It may be obtained by multiplying an expression e' of degree strictly less than k and a power of the form $x^i y^j$ of total degree no larger than k, with:

$$i \geq 0, \quad j \geq 0, \quad \text{and} \quad i + j \neq 0,$$

and such that

- for each monomial $a_{m,n} \cdot x^{i'} y^{j'}$ of the expression e', the expression $\left(a_{m,n} \cdot x^{i'} y^{j'}\right) \cdot x^i y^j$ is valid, that is,
$$m \geq i + i' \quad \text{and} \quad n \geq j + j'. \tag{6.4}$$
- at least one monomial $a_{m,n} \cdot x^{i'} y^{j'}$ of this valid expression e satisfies:
$$i + i' + j + j' = k. \tag{6.5}$$

Considering the polynomial $a(x, y)$ in Example 6.1, the condition in (6.4) enables to avoid expressions with monomials of the form $(a_{1,0} \cdot x) \cdot x$, for example.

Rule R3 for building the expressions by addition. Let e be an expression of total degree k, that belongs to $\mathrm{E}^{(k)}$. It may be obtained by adding two expressions e_1 and e_2 of degrees $k_1 = k$ and $k_2 \leq k$, respectively, such that:

- each coefficient of the polynomial $a(x, y)$ appears at most once in e,
- the expression e does not contain two coefficients having exactly the same degrees in x and y,
- and for all powers $x^i y^j$, with $i, j < k$,
$$e_1 \cdot x^i y^j \quad \text{is valid if and only if} \quad e_2 \cdot x^i y^j \quad \text{is valid.} \tag{6.6}$$

The third condition in (6.6) enables to avoid expressions where the coefficients have not to be multiplied by the same power to be involved in the evaluation of the polynomial $a(x, y)$. To illustrate this, consider the polynomial $a(x, y)$ in Example 6.1: the condition in (6.6) enables not to generate the expression $a_{0,0} + a_{2,0} \cdot x$.

Let us now see in the remainder of the section the general algorithm, and those used for implementing each of these building rules.

6.1.3 Description of the algorithm and implementation of the building rules

From the definition of the building rules in Section 6.1.2 above, and since at step 0 (initialization) we have:

$$\mathrm{E}^{(0)} = \bigcup_{i=0}^{n_x} \bigcup_{j=0}^{n_y} \{a_{i,j}\} \quad \text{and} \quad \mathrm{P}^{(1)} = \{x, y\}, \tag{6.7}$$

it follows that the computation of all the parenthesizations of $a(x, y)$ can be done iteratively. More precisely, for $k \geq 1$, step k consists in:

1. applying the rule R1 for computing $\mathrm{P}^{(k+1)}$, the powers of degree $k+1$;
2. applying the rules R2 and then R3 for computing $\mathrm{E}^{(k)}$.

From that statement, we can derive the general algorithm presented in Algorithm 6.1 below. We observe that the algorithm iterates $n-1$ times, and to show that the algorithm terminates, it suffices to show that each building rule implementation terminates.

Input: degrees n_x, n_y of the polynomial $a(x, y)$ in (6.1)
Output: $\mathrm{E}^{(n)}$, with $n = n_x + n_y$

```
   /* Initialization of P^(1) and E^(0)                                   */
1  P^(1) ← {x, y};
2  E^(0) ← ⋃_{i=0}^{n_x} ⋃_{j=0}^{n_y} {a_{i,j}};

3  for k ← 1 to n − 1 do                          /* for each degree 1 to n − 1 */
4  |  P^(k+1) ← BuildingRuleR1(k + 1, P^(1), ..., P^(k));          /* Rule1 */
5  |  E^(k) ← BuildingRuleR2(k, P^(1), ..., P^(k), E^(0), ..., E^(k−1));   /* Rule2 */
6  |  E^(k) ← BuildingRuleR3(k, P^(1), ..., P^(k), E^(0), ..., E^(k));     /* Rule3 */
7  end
   /* Last step:  computation of E^(n)                                    */
8  E^(n) ← BuildingRuleR2(n, P^(1), ..., P^(n), E^(0), ..., E^(n−1));     /* Rule2 */
9  E^(n) ← BuildingRuleR3(n, P^(1), ..., P^(n), E^(0), ..., E^(n));       /* Rule3 */
10 return E^(n);
```

Algorithm 6.1: Bivariate(n_x, n_y).

Remark that at the last step n, we do not need to execute the rule R1, since it creates the powers of degree $n+1$, and no such expressions are needed for evaluating the polynomial $a(x, y)$ of total degree n.

Let us now detail the implementation of each of the three building rules described in Section 6.1.2 above. For each implementation, the function `isvalid` check if the conditions required by the building rules in Section 6.1.2 are satisfied. However, their implementation is not given here.

Rules R1 and R2 for building powers and expressions by multiplications. From the definition of the building rules in Section 6.1.2, Algorithms 6.2 and 6.3 below immediately follow. Remark that both algorithms iterate a finite number of times, and at each iteration combine once a finite number of elements. So, we conclude that both algorithms terminate.

Input: step k and power sets $\mathrm{P}^{(1)}, \ldots, \mathrm{P}^{(k-1)}$

Output: $\mathrm{P}^{(k)}$, that contains the powers of total degree k

```
   // Initialization of P^(k)
1  P^(k) ← {};
   /* Consider each pair of P^(k') × P^(k-k'), with k' ∈ {1,...,⌊k/2⌋}        */
2  for k' ← 1 to ⌊k/2⌋ do
3      for i ← 1 to length(P^(k')) do          /* for each power p1 of P^(k') */
4          j ← 1;
5          if k' = k/2 then j ← i;
6          while j ≤ length(P^(k-k')) do /* for each power p2 of P^(k-k'), p2 ≠ p1 */
7              e ← P^(k')[i] × P^(k-k')[j];
8              if isvalid(e) then                    /* if e = p1 · p2 is valid */
9                  P^(k) ← P^(k) ∪ {e};
10             end
11             j ← j + 1;
12         end
13     end
14 end
15 return P^(k);
```

Algorithm 6.2: BuildingRuleR1($k, \mathrm{P}^{(1)}, \ldots, \mathrm{P}^{(k-1)}$).

Rule R3 for building expressions by additions. Using the building rule R3, we can write Algorithm 6.4, which computes the expressions of degree k by adding expressions of degrees at most k together. The correctness of this algorithm is less immediate than the two previous ones. So, let us give some key elements about it. To do so, let us consider step k, and the ith application of rule R3.

- First, from line 5 to line 17, the algorithm consists in adding the expressions of degree k created by the $(i-1)$st application of R3 (or by the application of R2, if $i = 1$) with all the expressions of degree $k' \leq k$. This yields a set tmp$[i]$ of expressions of degree k.
- Then, from line 18 to line 29, the algorithm consists in adding the expressions of degree k of tmp$[i-1]$ (created by the $(i-1)$st application of R3) to the expressions of degree k of tmp$[2], \ldots,$ tmp$[i-2]$.

Let MinNbrAdd$_{k,i}$ be the minimum number of additions involved in the expressions of degree k after the ith application of rule R3. It follows that MinNbrAdd$_{k,i+1}$ = 1 + MinNbrAdd$_{k,i}$. Now, the number of additions for evaluating a given polynomial of degree n remains constant whatever the parenthesization is. Hence at a given step k, the number of applications of rule R3 is finite and it follows that Algorithm 6.4 terminates.

At step n, the set $\mathrm{E}^{(n)}$ contains all the *valid* expressions of total degree n that can be used for evaluating the polynomial $a(x, y)$ in (6.1). It remains finally to extract from that set the expressions that represent exactly the polynomial $a(x, y)$ to be evaluated.

Remark 6.1. *Note that a proof of that no parenthesization has been missed by our approach still remains to be done.*

Input: step k, power sets $\mathrm{P}^{(1)}, \ldots, \mathrm{P}^{(k)}$, and expression sets $\mathrm{E}^{(0)}, \ldots, \mathrm{E}^{(k-1)}$

Output: $\mathrm{E}^{(k)}$

```
// Initialization of E^(k)
E^(k) ← {};
/* Consider each pair of E^(k') × P^(k-k'), with k' ∈ {0,...,k-1}   */
for k' ← 0 to k − 1 do
    for i ← 1 to length(E^(k')) do        /* for each expression e' of E^(k') */
        for j ← 1 to length(P^(k-k')) do   /* for each power p of P^(k-k') */
            e ← E^(k)[i] × P^(k-k')[j];
            if isvalid(e) then             /* if e = e' · p is valid */
                E^(k) ← E^(k) ∪ {e};
            end
        end
    end
end
return E^(k);
```

Algorithm 6.3: BuildingRuleR2($k, \mathrm{P}^{(1)}, \ldots, \mathrm{P}^{(k)}, \mathrm{E}^{(0)}, \ldots, \mathrm{E}^{(k-1)}$).

Let us now see an example of execution.

6.1.4 Example: evaluation of general bivariate degree-2 polynomials

This section is devoted to give an example of execution of Algorithm 6.1. Let us consider the general degree-2 bivariate polynomial $a(x, y)$ defined as follows:

$$a(x, y) = a_{0,0} + a_{1,0} \cdot x + a_{0,1} \cdot y + a_{1,1} \cdot x \cdot y. \tag{6.8}$$

Here, we will explain how to determine the result of each step. Assume first that the algorithm is initialized with the following sets, as in (6.7):

$$\mathrm{E}^{(0)} = \{a_{0,0}, a_{1,0}, a_{0,1}, a_{1,1}\} \quad \text{and} \quad \mathrm{P}^{(1)} = \{x, y\}. \tag{6.9}$$

Execution of step 1

Recall that step 1 consists in applying all the building rules, presented in Section 6.1.2, on the elements built in the previous steps, here $\mathrm{E}^{(0)}$ and $\mathrm{P}^{(1)}$ in (6.9), and computing all the expressions of total degree 1.

- Using $\mathrm{P}^{(1)}$ in (6.9), it follows that applying rule R1 gives the set $\mathrm{P}^{(2)}$ of powers of degree 2, defined as:

  $$\mathrm{P}^{(2)} = \{x \cdot y\}. \tag{6.10}$$

  Remark that these expressions will not be used in step 1, but in step 2. Hence, we could have applied this rule at the end of the step 1.

Input: step k, power sets $\mathrm{P}^{(1)}, \ldots, \mathrm{P}^{(k)}$, expression sets $\mathrm{E}^{(0)}, \ldots, \mathrm{E}^{(k-1)}$, and a subset of $\mathrm{E}^{(k-1)}$

Output: expression set $\mathrm{E}^{(k)}$

```
// Initialization
idx ← 1;
tmp[idx] ← E^(k);
repeat
    idx ← idx + 1;
    /* Consider each pair of E^(k) × E^(k'), with k' ∈ {0,...,k}                */
    for k' ← 0 to k do
        for i ← 1 to length(E^(k')) do      /* for each expression e_2 of E^(k') */
            j ← 1;
            if (idx = 2) ∧ (k' = k) then j ← i + 1;
            while j ≤ length(tmp[idx − 1]) do /* for each expression e_1 of E^(k),
            with e_1 ≠ e_2 */
                e ← E^(k')[i] + tmp[idx − 1][j];
                if isvalid(e) then                    /* if e = e_1 + e_2 is valid */
                    tmp[idx] ← tmp[idx] ∪ {e};
                end
                j ← j + 1;
            end
        end
    end
    for k' ← 2 to idx − 1 do
        for i ← 1 to length(tmp[k']) do     /* for each expression of degree k
        previously created by R3 */
            j ← 1;
            if k = idx − 1 then j ← i + 1;
            while j ≤ length(tmp[idx − 1]) do /* for each expression of degree
            k just created by R3 */
                e ← tmp[k'][i] + tmp[idx − 1][j];
                if isvalid(e) then
                    tmp[idx] ← tmp[idx] ∪ {e};
                end
            end
        end
    end
until length(tmp[idx]) ≠ 0 ;
for i ← 2 to idx − 1 do    /* concatenation of the expressions created.  */
    E^(k) ← E^(k) ∪ tmp[i];
end
return E^(k);
```

Algorithm 6.4: BuildingRuleR3($k, \mathrm{P}^{(1)}, \ldots, \mathrm{P}^{(k)}, \mathrm{E}^{(0)}, \ldots, \mathrm{E}^{(k)}$).

- Then, applying rule R2 consists in considering each pair of expressions $(e, p) \in \mathrm{E}^{(0)} \times \mathrm{P}^{(1)}$, and deducing all the valid resulting expressions, which are:

$$\{a_{1,0} \cdot x, a_{0,1} \cdot y, a_{1,1} \cdot x, a_{1,1} \cdot y\}. \tag{6.11}$$

  Remark here, that the expressions $a_{0,0} \cdot x$, $a_{0,0} \cdot y$, $a_{1,0} \cdot y$ and $a_{0,1} \cdot x$ are not valid, since they cannot be used for evaluating the polynomial $a(x, y)$ in (6.8).

- Finally, it remains to apply the rule R3, that consists in condering each pair of expressions (e_1, e_2) with e_1 an expression of degree 1 in (6.11) and e_2 a coefficient in (6.9) or an expression in (6.11), and computing all the valid expressions. It follows that the resulting expressions are:

$$\{a_{0,0} + a_{1,0} \cdot x, a_{0,0} + a_{0,1} \cdot y, a_{0,1} + a_{1,1} \cdot x, a_{1,0} + a_{1,1} \cdot y\} \quad \text{and} \quad \{a_{1,0} \cdot x + a_{0,1} \cdot y\}. \tag{6.12}$$

  Remark that the rule R3 takes expressions of degree at most 1 for creating expressions of degree 1. Hence, we have to apply this rule R3 to the new expressions in (6.12). Thus, let us consider each pair of expressions (e_1, e_2) with e_1 an expression in (6.12), and e_2 a coefficient in (6.9), an expression in (6.11), or in (6.12). By iterating the process until no more expression is created, we obtained the set $\mathrm{E}^{(1)}$ defined as follows

$$\begin{aligned} \mathrm{E}^{(1)} \quad = \quad & \{\ a_{1,0} \cdot x, a_{0,1} \cdot y, a_{1,1} \cdot x, a_{1,1} \cdot y, \\ & a_{0,0} + a_{1,0} \cdot x, a_{0,0} + a_{0,1} \cdot y, a_{0,1} + a_{1,1} \cdot x, a_{1,0} + a_{1,1} \cdot y, \\ & a_{1,0} \cdot x + a_{0,1} \cdot y, a_{0,0} + (a_{1,0} \cdot x + a_{0,1} \cdot y)\ \}. \end{aligned} \tag{6.13}$$

Let us now see how to build from these expressions the ones of degree 2, that evaluate the polynomial $a(x, y)$ in (6.8).

Execution of step 2

At step 2, rule R1 is not used. Indeed its application on the elements of $\mathrm{P}^{(1)}$ and $\mathrm{P}^{(2)}$ would have produced the empty set, since no powers of degree 3 can be used for evaluating the degree-2 polynomial in (6.8).

Hence, let us first see the result of the application of rule R2. To do so, we consider each pair (e, p) where e belongs to $\mathrm{E}^{(0)}$ in (6.9) and p to $\mathrm{P}^{(2)}$ in (6.10), or where e belongs to $\mathrm{E}^{(1)}$ in (6.13) and p to $\mathrm{P}^{(1)}$ in (6.9). Then multiplying e by p leads to an expression of degree 2. After eliminating the non-valid expressions thus produced, we finally obtain:

$$\{a_{1,1} \cdot (x \cdot y)\} \quad \text{and} \quad \{(a_{1,1} \cdot y) \cdot x, (a_{1,0} + a_{1,1} \cdot y) \cdot x, (a_{1,1} \cdot x) \cdot y, (a_{0,1} + a_{1,1} \cdot x) \cdot y\}. \tag{6.14}$$

Finally, it remains to apply the rule R3 to the expressions in (6.14) together with those built in the previous steps. To illustrate that, let us see how to obtain the following parenthesization:

$$a_{0,0} + \Big[(a_{0,1} \cdot y) + \Big((a_{1,0} \cdot x) + \big(a_{1,1} \cdot (x \cdot y)\big)\Big)\Big]. \tag{6.15}$$

A first application of rule R3 to the elements of (6.14) together with those of $\mathrm{E}^{(1)}$ in (6.13) leads to the following expression:

$$(a_{1,0} \cdot x) + \big(a_{1,1} \cdot (x \cdot y)\big)$$

Applying a second time rule R3 to these same elements gives:

$$\Big[(a_{0,1} \cdot y) + \Big((a_{1,0} \cdot x) + \big(a_{1,1} \cdot (x \cdot y)\big)\Big)\Big]. \tag{6.16}$$

A last application of rule R3 to (6.16) and the elements of $\mathrm{E}^{(0)}$ yields the parenthesizations in (6.15). Actually by iterating until no more evaluation scheme is computed, we get 51 evaluation parenthesizations, shown in Listing 6.1 below. (Remark that in Listing 6.1, the coefficients are presented under the syntax of CGPE: they are numbered from 0 to 3, so that the polynomial to be evaluated is $a_0 + a_1 \cdot x + a_2 \cdot y + a_3 \cdot xy$.)

We observe that since $a_{n_x,n_y} \neq 0$ in (6.1), the application of rule R3 cannot build expression by adding two expressions having the same total degree. Hence during the last step, lines 18 to 29 of Algorithm 6.4 cannot create any valid expressions and thus, can be skipped.

6.1.5 Improvement for some "special" polynomials

Until now, we have detailed the algorithm used for generating all the parenthesizations for evaluating general bivariate polynomials. Let us now consider polynomials that contain only one coefficient of each degree, that is, whose all monomials $a_{i,j}x^iy^j$ satisfy the following property:

$$\text{if} \quad a_{i,j}x^iy^j \neq a_{i',j'}x^{i'}y^{j'} \quad \text{then} \quad i+j \neq i'+j'.$$

This is in particular the case for univariate polynomials or for the particular bivariate polynomials used in Part I for the implementation of roots and division, for example. Let us now detail two improvements that can be done for generating all the parenthesizations of such polynomials.

First improvement. In this case, we may remark that in Algorithm 6.4, we cannot build any expressions by adding two expressions of the same degree. Hence, a first improvement consists in simplifying Algorithm 6.4 by considering only pairs of expressions (e_1, e_2) of distinct degrees. This can mainly be done by considering $k' \neq k$ between lines 5 and17 and by removing the lines 18 to 29. (This new version will be called *UnivariateLike*, later in this section.)

Second improvement To illustrate the second improvement, let us consider degree-3 univariate polynomials:

$$a(x) = a_{0,0} + a_{1,0} \cdot x + a_{2,0} \cdot x^2 + a_{3,0} \cdot x^3.$$

Consider step 3 of Algorithm 6.1, and that we have already applied rules R1 and R2. Hence, we have $\mathrm{E}^{(k)}$, for $k \in \{0, \ldots, 3\}$, where $\mathrm{E}^{(3)}$ contains a subset of all the parenthesizations for evaluating degree-3 expressions. By denoting e_i an expression of degree i or equal to 0, that is, $e_i \in \mathrm{E}^{(i)} \cup \{0\}$, we observe that

$$a(x) = e_0 + e_1 + e_2 + e_3.$$

For example, we could have:

$$a(x) = (a_{0,0} + a_{1,0} \cdot x) + (x \cdot x) \cdot (a_{2,0} + a_{3,\cdot}x),$$

and

$$e_0 = 0, \quad e_1 = a_{0,0} + a_{1,0} \cdot x, \quad e_2 = 0, \quad \text{and} \quad e_3 = (x \cdot x) \cdot (a_{2,0} + a_{3,\cdot}x).$$

It remains now to find all the parenthesizations. Since addition is commutative, we observe that the polynomial $a(x)$ can be defined by considering each permutation of the elements $\{e_0, e_1, e_2\}$, as follows:

$$a(x) = \begin{cases} [(e_3 + e_2) + e_1] + e_0 \\ [(e_3 + e_1) + e_2] + e_0 \\ [(e_3 + e_2) + e_0] + e_1 \\ [(e_3 + e_0) + e_2] + e_1 \\ [(e_3 + e_1) + e_0] + e_2 \\ [(e_3 + e_0) + e_1] + e_2 \end{cases}$$

```
1  a(x,y) = ((a0+((x*a1)+(y*a2)))+(x*(y*a3)))
2  a(x,y) = ((a0+((x*a1)+(y*a2)))+(y*(x*a3)))
3  a(x,y) = ((a0+((x*a1)+(y*a2)))+((x*y)*a3))
4  a(x,y) = (((x*a1)+(a0+(y*a2)))+(x*(y*a3)))
5  a(x,y) = (((x*a1)+(a0+(y*a2)))+(y*(x*a3)))
6  a(x,y) = (((x*a1)+(a0+(y*a2)))+((x*y)*a3))
7  a(x,y) = (((y*a2)+(a0+(x*a1)))+(x*(y*a3)))
8  a(x,y) = (((y*a2)+(a0+(x*a1)))+(y*(x*a3)))
9  a(x,y) = (((y*a2)+(a0+(x*a1)))+((x*y)*a3))
10 a(x,y) = ((a0+(x*a1))+(y*(a2+(x*a3))))
11 a(x,y) = ((a0+(y*a2))+(x*(a1+(y*a3))))
12 a(x,y) = (a0+(((x*a1)+(y*a2))+(x*(y*a3))))
13 a(x,y) = (a0+(((x*a1)+(y*a2))+(y*(x*a3))))
14 a(x,y) = (a0+(((x*a1)+(y*a2))+((x*y)*a3)))
15 a(x,y) = ((x*a1)+((a0+(y*a2))+(x*(y*a3))))
16 a(x,y) = ((x*a1)+((a0+(y*a2))+(y*(x*a3))))
17 a(x,y) = ((x*a1)+((a0+(y*a2))+((x*y)*a3)))
18 a(x,y) = ((y*a2)+((a0+(x*a1))+(x*(y*a3))))
19 a(x,y) = ((y*a2)+((a0+(x*a1))+(y*(x*a3))))
20 a(x,y) = ((y*a2)+((a0+(x*a1))+((x*y)*a3)))
21 a(x,y) = ((a0+(x*a1))+((y*a2)+(x*(y*a3))))
22 a(x,y) = ((a0+(x*a1))+((y*a2)+(y*(x*a3))))
23 a(x,y) = ((a0+(x*a1))+((y*a2)+((x*y)*a3)))
24 a(x,y) = ((a0+(y*a2))+((x*a1)+(x*(y*a3))))
25 a(x,y) = ((a0+(y*a2))+((x*a1)+(y*(x*a3))))
26 a(x,y) = ((a0+(y*a2))+((x*a1)+((x*y)*a3)))
27 a(x,y) = (((x*a1)+(y*a2))+(a0+(x*(y*a3))))
28 a(x,y) = (((x*a1)+(y*a2))+(a0+(y*(x*a3))))
29 a(x,y) = (((x*a1)+(y*a2))+(a0+((x*y)*a3)))
30 a(x,y) = (a0+((x*a1)+(y*(a2+(x*a3)))))
31 a(x,y) = (a0+((y*a2)+(x*(a1+(y*a3)))))
32 a(x,y) = ((x*a1)+(a0+(y*(a2+(x*a3)))))
33 a(x,y) = ((y*a2)+(a0+(x*(a1+(y*a3)))))
34 a(x,y) = (a0+((x*a1)+((y*a2)+(x*(y*a3)))))
35 a(x,y) = (a0+((x*a1)+((y*a2)+(y*(x*a3)))))
36 a(x,y) = (a0+((x*a1)+((y*a2)+((x*y)*a3))))
37 a(x,y) = (a0+((y*a2)+((x*a1)+(x*(y*a3)))))
38 a(x,y) = (a0+((y*a2)+((x*a1)+(y*(x*a3)))))
39 a(x,y) = (a0+((y*a2)+((x*a1)+((x*y)*a3))))
40 a(x,y) = ((x*a1)+(a0+((y*a2)+(x*(y*a3)))))
41 a(x,y) = ((x*a1)+(a0+((y*a2)+(y*(x*a3)))))
42 a(x,y) = ((x*a1)+(a0+((y*a2)+((x*y)*a3))))
43 a(x,y) = ((x*a1)+((y*a2)+(a0+(x*(y*a3)))))
44 a(x,y) = ((x*a1)+((y*a2)+(a0+(y*(x*a3)))))
45 a(x,y) = ((x*a1)+((y*a2)+(a0+((x*y)*a3))))
46 a(x,y) = ((y*a2)+(a0+((x*a1)+(x*(y*a3)))))
47 a(x,y) = ((y*a2)+(a0+((x*a1)+(y*(x*a3)))))
48 a(x,y) = ((y*a2)+(a0+((x*a1)+((x*y)*a3))))
49 a(x,y) = ((y*a2)+((x*a1)+(a0+(x*(y*a3)))))
50 a(x,y) = ((y*a2)+((x*a1)+(a0+(y*(x*a3)))))
51 a(x,y) = ((y*a2)+((x*a1)+(a0+((x*y)*a3))))
```

Listing 6.1: All the evaluation parenthesizations for evaluating a degree-2 bivariate polynomial.

This second improvement is called *UnivariateLike-Optim*. Its interest relies on the fact that, at step 3, once we have applied rule R2, each permutation of $\{e_0, e_1, e_2\}$ can be considered separately. This improvement has been implemented in CGPE, where, each permutation of $\{e_0, e_1, e_2\}$ is considered iteratively one after the other.

Moreover, the computation of all the parenthesizations for evaluating the degree-3 univariate polynomial $a(x)$ can be done in parallel, as shown in Figure 6.3.

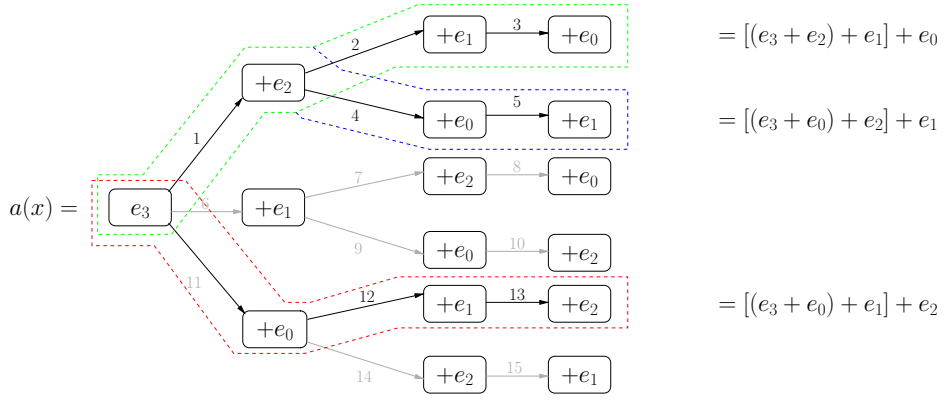


Figure 6.3: Strategy for parallel generation of parenthesizations for evaluating univariate degree-3 polynomials.

Remark, in Figure 6.3, that each node of the tree could be launched on a particular processor, which would lead to a parallel version of the algorithm. Unfortunately, this version has still not been implemented.

6.1.6 Numerical results

This algorithm has been integrated into the software environment CGPE [Rev]. This section presents some numerical results, obtained using this tool, for evaluating various kinds of polynomials.

Bivariate and univariate polynomials

Let $a(x, y)$ be a bivariate polynomial, defined as in (6.1), with n_x and n_y its degrees in x and y, respectively. Table 6.1 below gives the numbers of all possible parenthesizations for evaluating the polynomial $a(x, y)$, for some degrees (n_x, n_y). The underlined numbers indicate that these parenthesizations have not been currently generated but only counted, using what is done in [Mou09]. We can observe that even for small degree, the numbers of possible parenthesizations may be extremely large, and consequently, the timings for generating all these parenthesizations as well. For example, generating the 2334244 parenthesizations for the evaluation of degree-5 univariate polynomials takes about 1 hour 15m on a 2,4 Ghz core, using *UnivariateLike* algorithm, while

	$n_x = 1$	$n_x = 2$	$n_x = 3$	$n_x = 4$	$n_x = 5$	$n_x = 6$
$n_y = 0$	1	7	163	11602	2334244	1304066578
$n_y = 1$	51	67467	1133220387	207905478247998	-	
$n_y = 2$	67467	106191222651	10139277122276921118	-	-	

Table 6.1: Number of generated parenthesizations for evaluating a bivariate polynomial.

generating the 67467 parenthesizations for the evaluation of degree-$(2,1)$ bivariate polynomial takes about 30 seconds.

Impact of the *UnivariateLike-Optim* algorithm. Remark, from now, that by using the *UnivariateLike-Optim* algorithm, the timings for generating all the 2334244 parenthesizations of a degree-5 univariate polynomial falls to about 4 minutes, that is, in this case we obtain a speed up by a factor larger than 18 compared to the *UnivariateLike* algorithm. Hereafter, we will always use this version of the algorithm for our "special" polynomials, since it is much faster.

Some particular bivariate polynomials

In our context, the polynomials to be evaluated are of the form $P(s,t) = 2^{-p-1} + s \cdot a(t)$, with $a(t)$ a univariate polynomial. They can be seen as general bivariate polynomial as in (6.1) with $a(t)$ a univariate degree-$n_t x$ polynomial, $n_s = 1$, and some coefficients equal to 0. Hence Algorithm 6.1 can be used by simply removing all the zero coefficients from the initial set $\mathrm{E}^{(0)}$.

	$n_t = 1$	$n_t = 2$	$n_t = 3$	$n_t = 4$	$n_t = 5$	$n_t = 6$
$P(s,t) = 2^{-p-1} + s \cdot a(t)$	10	481	88384	-	-	-

Table 6.2: Number of generated parenthesizations for evaluating particular bivariate polynomials.

Table 6.2 gives the number of generated parenthesizations for the evaluation of the particular bivariate polynomial $P(s,t) = 2^{-p-1} + s \cdot a(t)$, with $a(t)$ a degree-n_t univariate polynomial. The generation of the 88384 parenthesizations for $n_t = 3$ is done within a few seconds. But those for $n_t = 4$ could not have been generated in a reasonable amount of time.

Anyway, in our context, we do not want to generate all the parenthesizations, but just those having the lowest latency, as will be detailed in Section 6.2.

What about the latency of evaluation?

Assuming that we want to integrate these algorithms into an automatic process for generating efficient parenthesizations, the problem is that we cannot generate all the parenthesizations in a faster way, especially for higher degrees, to choose the best one. So, let us see now the number of evaluation trees having the minimal latency. Here, we present some numerical results in terms of numbers of parenthesizations. The way used for generating them, especially to determine this latency, or at least a lower bound, is described in the next section. Tables 6.3 and 6.4 give for some degrees, the numbers of parenthesizations having minimal latency. (This minimal latency indicated between brackets). Here and hereafter, all the latencies are given for the ST231 processor, with an addition in 1 cycle and a pipelined multiplication in 3 cycles.

	$n_x = 1$	$n_x = 2$	$n_x = 3$	$n_x = 4$	$n_x = 5$	$n_x = 6$	$n_x = 7$
$n_y = 0$ [minimal latency]	1 [4]	2 [7]	12 [8]	187 [10]	36 [10]	9854 [11]	612 [11]
$n_y = 1$ [minimal latency]	9 [7]	129 [8]	135974 [10]	-	-	-	-

Table 6.3: Number of parenthesizations of minimal latency (between brackets) for evaluating a bivariate polynomials.

All these parenthesizations have been generated using CGPE as well. Let us make the following remarks:

	$n_t = 1$	$n_t = 2$	$n_t = 3$	$n_t = 4$	$n_t = 5$	$n_t = 6$
$P(s,t) = 2^{-p-1} + s \cdot a(t)$ [min. lat.]	3 [7]	33 [8]	1208 [10]	99 [10]	447803 [11]	10494 [11]

Table 6.4: Number of parenthesizations of minimal latency (between brackets) for evaluating some special bivariate polynomials.

- The generation of the 612 parenthesizations for evaluating degree-7 univariate polynomials in 11 cycles have been done in about 1s.
- The 447803 parenthesizations of latency 11 cycles for degree-(5,1) particular bivariate polynomials $P(s,t) = 2^{-p-1} + s \cdot a(t)$ have been generated in about 25s.
- Finally, the generation of the 10494 parenthesizations of latency 11 for evaluating particular degree-(6,1) bivariate polynomials $P(s,t) = 2^{-p-1} + s \cdot a(t)$ has been done in approximatively 2m30s.

These results point out the interest of finding a minimal latency, and generating only the parenthesizations that have a latency as close as possible to it. Section 6.2 below details the way used to determine such a minimal target latency.

6.2 Computing parenthesizations of low evaluation latency

We have seen in the previous section that clearly the number of parenthesizations for evaluating a given polynomial is extremely large, even for univariate polynomials of small degree (≤ 5). However, we observe in Table 6.3 above that, for example, among the 2334244 parenthesizations evaluating a degree-5 univariate polynomial, only 36 are of minimal latency of 10 cycles.

Basically in most cases, we want to keep the parenthesizations that reduce the evaluation latency on unbounded parallelism, and more particularly we do not want to compute the other ones. Hence if we manage to determine in a faster way the minimal latency for the evaluation of $a(x,y)$, at each step of the computation of the parenthesizations in Algorithm 6.1, we will be able to keep only the ones having a latency no larger than this bound, and thus to converge quickly towards some parenthesizations having minimal latency. Determining such a minimal latency may be costly, and we prefer here to determine a lower bound on this latency. This approach is heuristic, and if no evaluation scheme satisfying this target latency has been computed at the end of the process, we increment it by one and restart the computation.

Therefore this section presents the method used for determining *a priori* a *target latency*, for evaluating the bivariate polynomial $a(x,y)$ defined in (6.1).

6.2.1 Definition of the *target latency*

Assume now that the last operation of the evaluation of the polynomial $a(x,y)$ is an addition ($a_{0,0} \neq 0$). Since this evaluation consists at least in evaluating

$$a_{0,0} + a_{n_x,n_y} x^{n_x} y^{n_y},$$

this target latency can be obtained as the latency of the evaluation of the leading monomial plus the latency of the last addition. In the following of this section, we will denote by A and M the latencies of addition and multiplication, respectively. For example, on the ST231 processor, we have $A = 1$ and $M = 3$.

Example 6.3. *Let $a(x, y)$ be a degree-2 polynomial defined as follows*

$$a(x, y) = a_{0,0} + a_{1,0} \cdot x + a_{0,1} \cdot y + a_{1,1} \cdot x \cdot y.$$

Any program that evaluates $a(x, y)$ is also able to evaluate $a_{0,0} + a_{1,1} \cdot x \cdot y$. We know that there exists three parenthesizations for evaluating $a_{1,1} \cdot x \cdot y$, which are

$$a_{1,1} \cdot (x \cdot y), \quad (a_{1,1} \cdot x) \cdot y, \quad \text{and} \quad (a_{1,1} \cdot y) \cdot x.$$

All these parenthesizations have a latency of $2 \times M = 6$ cycles. We deduce the following target latency $\tau = 2M + A$, which on ST231, is:

$$\tau = 2 \times 3 + 1 = 7 \text{ cycles}.$$

For the sake of clarity, we denote by a' the leading coefficient of the polynomial $a(x, y)$, that is,

$$a' = a_{n_x, n_y}.$$

It remains now to determine a lower bound on the evaluation of the leading monomial.

A first approach consists in assuming that evaluating $a' x^{n_x} y^{n_y}$ in unbounded parallelism is at least as general as evaluating $x^{n_x+n_y+1}$. Since evaluating $x^{n_x+n_y+1}$ requires at least $\lceil \log_2(n_x + n_y + 1) \rceil$ successive multiplications [Knu98, §4.6.3], we deduce the following *static* target latency:

$$\tau_{\text{static}} = \lceil \log_2(n_x + n_y + 1) \rceil \times M + A. \tag{6.17}$$

This first lower bound is static, and does not take into account the inner structure of the problem. Let us now have a look at the determination of a dynamic target latency.

6.2.2 Determination of a *dynamic* target latency

This section presents the method we have implemented to determine a *dynamic* target latency, that takes into account the specification of the problem. Indeed, we have seen in Part I that during the implementation of a mathematical function, one of the following cases may occur:

- One of the indeterminates may be obtained a few cycles later than the other one (in the case of bivariate polynomial evaluation, only): this is what we called the *delay* on one of the indeterminates;
- Some polynomial coefficients may be forced to be a power of 2 so that multiplications by these coefficients may be replaced by simple shifts on the targeted integer architecture.

Let us now see how to determine this latency. To do so, we denote by $AB(i, j)$ the minimal latency for evaluating a monomial $a' x^i y^j$. Hence, it follows that

$$\tau_{\text{dynamic}} = AB(n_x, n_y) + A. \tag{6.18}$$

In the remainder of this section, we denote by D_x (respectively D_y) the cost for obtaining the indeterminate x (respectively y). The delay of y with respect to x thus is equal to $D_y - D_x$. We also write M' for the latency of the multiplication by the leading coefficient a'. Let us now see first how to compute the latency for evaluating the expressions of the form x^{n_x} and $a' x^{n_x}$, then those to compute $x^{n_x} y^{n_y}$ and $a' x^{n_x} y^{n_y}$.

Minimal latency for the evaluation of x^{n_x} and $a'x^{n_x}$

Following what is done in [Knu98, §4.6.3], we want first to compute the minimal latency for the evaluation of x^{n_x} (with $n_x \in \mathbb{N}$). Let $k \in \mathbb{N}$ and $0 \leq k \leq n_x$, such as

$$x^{n_x} = x^k \times x^{n_x-k}.$$

Since obviously $x^k \times x^{n_x-k} = x^{n_x-k} \times x^k$, we can restrict k to be in $\{1, 2, \ldots, \lfloor n_x/2 \rfloor\}$. The objective is to determine the value k for which the evaluation latency of x^{n_x} is minimal. To do so, let $U_x(n_x)$ be the minimal latency for the evaluation of x^{n_x} assuming that x is obtained after D_x cycles. Given n_x, one of the following cases may occur.

- If $n_x = 0$, then we have $U_x(n_x) = 0$.
- If $n_x = 1$, then we have $U_x(n_x) = D_x$, since it corresponds to the latency for obtaining the indeterminate x and no multiplication is required to compute $x^{n_x} = x$.
- Otherwise, given a value k, the lowest latency for evaluating $x^{n_x} = x^k \times x^{n_x-k}$ is defined as follows:
$$M + \max\left(U_x(k), U_x(n_x - k)\right).$$
as illustrated in Figure 6.4 below.

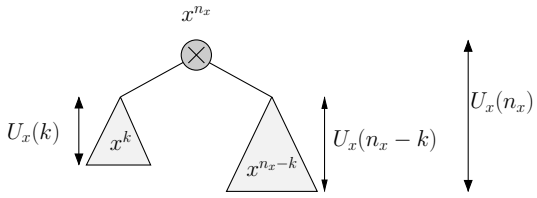


Figure 6.4: Latency for the evaluation of $x^{n_x} = x^k \times x^{n_x-k}$.

It follows that the minimal latency for evaluating x^{n_x} is defined as:

$$U_x(n_x) = \begin{cases} 0 & \text{if } n_x = 0, \\ D_x & \text{if } n_x = 1, \\ \min_{k=1,2,\ldots,\lfloor n_x/2 \rfloor} \left\{ M + \max\left(U_x(k), U_x(n-k)\right) \right\} & \text{otherwise.} \end{cases} \tag{6.19}$$

Using the same approach, we immmediately obtain a definition for $AU_x(n_x)$, the minimal latency for evaluating $a'x^{n_x}$:

$$AU_x(n_x) = \begin{cases} 0 & \text{if } n_x = 0, \\ M' + D_x & \text{if } n_x = 1, \\ \min_{k=0,\ldots,n_x-1} \left\{ M + \max\left(AU_x(k), U_x(n-k)\right) \right\} & \text{otherwise.} \end{cases} \tag{6.20}$$

Remark that $U_y(n_y)$ and $AU_y(n_y)$ can be defined in the same way.

Minimal latency for the evaluation of $x^{n_x}y^{n_y}$ and $a'x^{n_x}y^{n_y}$

In the same way as for the computation of the minimal latency for the evaluation of x^{n_x} in the previous paragraph, this goal is here to find a pair (k_x, k_y) such that

$$x^{n_x}y^{n_y} = x^{k_x}y^{k_y} \times x^{n_x-k_x}y^{n_y-k_y},$$

and so that the latency of the evaluation of $x^{n_x}y^{n_y}$ remains minimal. To do so, let us define $B(n_x, n_y)$ the minimal latency for the evaluation of $x^{n_x}y^{n_y}$ assuming that x and y are available after D_x and D_y cycles, respectively. Then, one of the following cases occurs.

- If $n_x = 0$ then $B(n_x, n_y) = U_y(n_y)$, since it amounts to evaluate y^{n_y}.
- If $n_y = 0$ then $B(n_x, n_y) = U_x(n_x)$, since it amounts to evaluate x^{n_x}.
- Otherwise, given the values k_x and k_y, the lowest latency for evaluating the powers $x^{n_x}y^{n_y}$ is defined as follows:

$$M + \max\big(B(k_x, k_y), B(n_x - k_x, n_y - k_y)\big),$$

with $k_x + k_y \neq 0$ and $k_x + k_y \neq n_x + n_y$.

Assume here that $k_x \in \{0, \ldots, n_x\}$ and $k_y \in \{0, \ldots, n_y\}$, with $k_x + k_y \neq 0$ and $k_x + k_y \neq n_x + n_y$. It follows that the minimal latency for evaluating the powers $x^{n_x}y^{n_y}$ is defined as:

$$B(n_x, n_y) = \begin{cases} U_y(n_y) & n_x = 0, \\ U_x(n_x) & n_y = 0, \\ \min_{k_x,k_y}\Big\{M + \max\big(B(k_x, k_y), B(n_x - k_x, n_y - k_y)\big)\Big\} & \text{otherwise.} \end{cases} \tag{6.21}$$

Finally, using (6.19), (6.20) and (6.21), we can deduce It follows finally Algorithm 6.5 for computing $AB(i, j)$ (the minimal latency for evaluating a monomial $a'x^iy^j$) and then deducing τ_{dynamic} in (6.18). For sake of efficiency, this algorithm is implemented using dynamic programming [CLR92].

6.2.3 Numerical results

We have seen in Section 6.1.6 the impact of the target latency on the number of generated parenthesizations and more particularly on the timing of the generation. Let us now have a look at the quality of the evaluation program generated using CGPE for evaluating polynomial of the form

$$P(s, t) = 2^{-p-1} + s \cdot a(t),$$

with $a(t)$ a degree-n_x polynomial. To fit the notation presented in the previous section, we assume here that $n_y = 1$. Table 6.5 below shows for various functions the degree of the polynomial approximant $a(t)$, the delay on the indeterminate s with respect to the indeterminate t (called D_s, assuming $D_t = 0$), and the latency on unbounded parallelism as well as the one on a simplified model of the ST231 processor (some details about this model are given in Section 6.4.2 below).

Impact of the delay of one of the indeterminates on the target latency

Table 6.6 below shows for various degrees n_x and various delays on the indeterminate s, the static target latency in (6.17) and the dynamic one computed as in (6.18). The interest of this table lies in the fact that it enables to decide if an improvement of the computation of the indeterminate s would lead to a possible improvement of the evaluation latency of the polynomial.

Consider for example the square root function presented in Chaper 3. As shown in Table 6.5 below, we have implemented this function with a degree-8 polynomial $a(t)$ and we know that the indeterminate s is obtained 3 cycles after t. Hence, we can see in Table 6.6 that even if we had a smaller delay on s, we could not have had a faster implementation. More particulalry, we could have a delay up to 6 cycles without any increase of the target latency. However, using the heuristic

Input: degrees n_x, n_y of the polynomial $a(x, y)$ in 6.1

Output: axy[n_x][n_y] = $AB(n_x, n_y)$, the minimal latency for evaluating a monomial $a'x^iy^j$

```
1  if axy[dx][dy] == -1 then      /* axy[n_x][n_y] have not been computed yet */
2      if dx == 0 then axy[n_x][n_y] = AUy(n_y);
3      else if dy == 0 then axy[n_x][n_y] = AUx(n_x);
4      else
           /* Special case:  k_x = 0 and k_y = 0                              */
5          minlatency ← M' + B(n_x, n_y);
6          for k_x ← 0 to n_x do
7              for k_y ← 0 to n_y do
8                  if k_x + k_y ≠ 0 and k_x + k_y ≠ n_x + n_y then
9                      minlatency
                       ← min(minlatency, M + max(AB(k_x, k_y), B(n_x − k_x, n_y − k_y)));
10                 end
11             end
12         end
13     axy[n_x][n_y] ← minlatency;
14 end
15 return axy[dx][dy];
```

Algorithm 6.5: AB(n_x, n_y), minimal latency for evaluating $a'x^{n_x}y^{n_y}$.

	$x^{1/2}$	$x^{-1/2}$	$x^{1/3}$	$x^{-1/3}$	$x^{1/4}$	$x^{-1/4}$	$x^{1/5}$	$x^{-1/5}$	x^{-1}	x/y
Degree (n_x,n_y)	(8,1)	(9,1)	(8,1)	(9,1)	(8,1)	(9*,1)	(8,1)	(8,1)	(10,0)	(10,1)
Delay D_s on the operand s (# cycles)	3	3	9	9	3	3	9	9	0	3
Latency on unbounded parallelism (# cycles)	13	13	16	16	13	13	16	16	13	14
Latency on ST231 (# cycles)	13	14	16	16	13	14	16	16	13	15

Table 6.5: Latency on unbounded parallelism and on ST231, for various functions of FLIP.
(* Note that here we consider a degree-9 polynomial approximant for the reciprocal fourth root, so that the validation can be done automatically.)

n_x	τ_{static}	τ_{dynamic} with $D_s = \dots$										
		0	1	2	3	4	5	6	7	8	9	10
1	7	7	7	7	7	8	9	10	11	12	13	14
2	7	7	8	9	10	10	10	10	11	12	13	14
3	10	10	10	10	10	10	10	10	11	12	13	14
4	10	10	10	10	10	11	12	13	13	13	13	14
5	10	10	10	10	10	11	12	13	13	13	13	14
6	10	10	11	12	13	13	13	13	13	13	13	14
7	13	13	13	13	13	13	13	13	13	13	13	14
8	13	13	13	13	13	13	13	13	14	15	16	16
9	13	13	13	13	13	13	13	13	14	15	16	16
10	13	13	13	13	13	13	13	13	14	15	16	16
11	13	13	13	13	13	13	13	13	14	15	16	16
12	13	13	13	13	13	14	15	16	16	16	16	16
13	13	13	13	13	13	14	15	16	16	16	16	16
14	13	13	14	15	16	16	16	16	16	16	16	16
15	16	16	16	16	16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16	16	16	16	16	17
17	16	16	16	16	16	16	16	16	16	16	16	17
18	16	16	16	16	16	16	16	16	16	16	16	17
19	16	16	16	16	16	16	16	16	16	16	16	17
20	16	16	16	16	16	16	16	16	16	16	16	17

Table 6.6: Target latency for various degrees n_x and various delays D_s.

of CGPE presented in Section 6.3, we cannot find any evaluation program in 13 cycles with a delay larger than 3.

As a second example, let us consider the cube root, implemented using a degree-9 polynomial approximant $a(t)$ and a delay on s of 9 cycles. From Table 6.6, we can observe that if we had s in 8 cycles instead of 9, we would have a target latency of 15 cycles. Furthermore, using CGPE, we have indeed found an evaluation program whose latency is of 15 cycles.

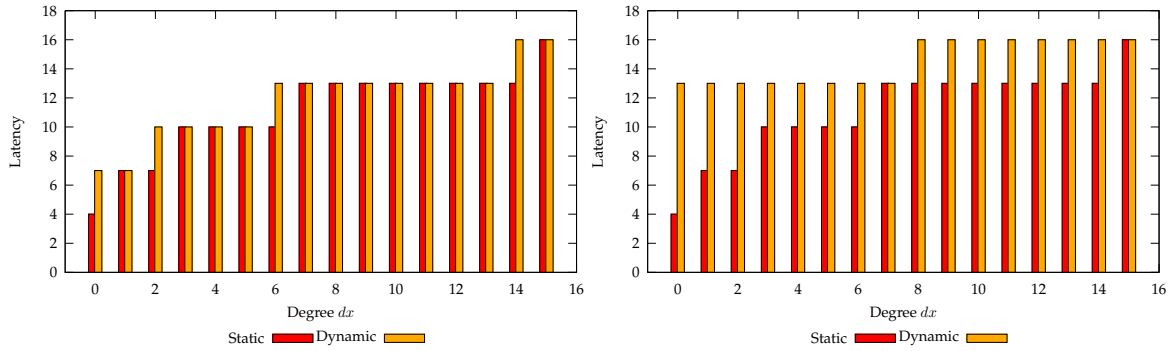
These examples point out the impact of the delay on the indeterminate s on the whole latency, and the interest of not neglecting its computation in the implementations presented in Part I.

Comparison of static and dynamic target latencies

Let us now observe if the evaluation program generated with CGPE for implementing the functions in Table 6.5 above are optimal in terms of latency of evaluation. We have computed the target latency for the implementation of the functions of Table 6.5 based on bivariate polynomial evaluation, which are displayed in Figure 6.5 below.

Let us first consider the square root function $x^{1/2}$ for which the degree in t of the approximant is 8. Using Figure 6.5(a), since for this function, s is obtained 3 cycles after the indeterminate t, we deduce that the polynomial approximant cannot be evaluated in fewer than 13 cycles. Hence, we conclude that, for the square root, the evaluation of the polynomial approximant is optimal on ST231.

In fact, the same conclusion follows for all the functions of Table 6.5 based on the evaluation of a bivariate polynomial, except for $x^{-1/2}$, $x^{-1/4}$, and x/y. Moreover, for the reciprocal function we can compute using $AU_x(10)$ in (6.20) a target latency of 13 cycles, that enables to conclude also for this function that the polynomial evaluation program is optimal.

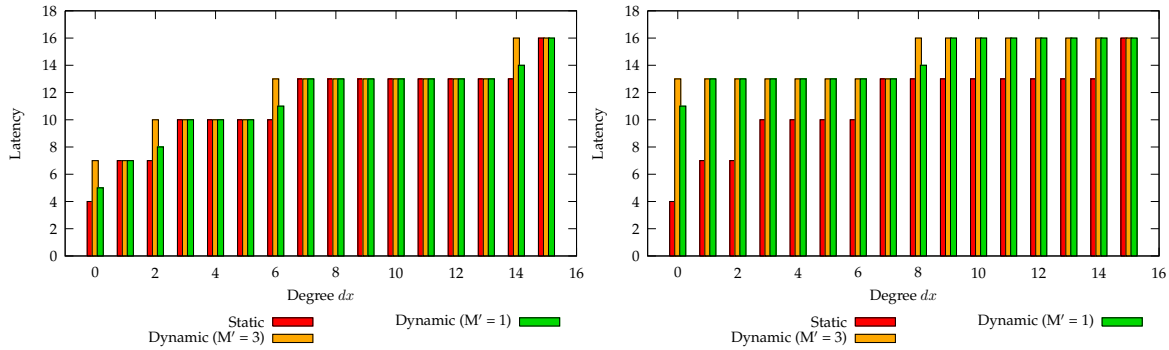


(a) With a delay $D_s = 3$. (b) With a delay $D_s = 9$.

Figure 6.5: Target latency for the evaluation of particular bivariate polynomials of the form $P(s,t) = 2^{-p-1} + s \cdot a(t)$.

Impact of the cost of the last multiplication

Until now, we have studied the impact on the target latency of the delay on s. In this third example, we will show the impact on this bound of the cost of the multiplication by the leading coefficient. Indeed, in Chapters 1 and 3, we have seen that forcing the leading coefficient to be a power of 2 may lead to a decrease of the size of the generated assembly code. But what about the evaluation latency? From the results of Figure 6.6, we can make the following remarks:



(a) With a delay $D_y = 3$. (b) With a delay $D_y = 9$.

Figure 6.6: Impact of forcing the leading coefficient to be a power of 2.

- From Figure 6.6(a), we can conclude that even if we force the leading coefficient of the degree-8 polynomial approximant for square root to be a power of 2, that is, $M' = 1$ (since it is equivalent to a simple shift), we cannot expect any improvement of the evaluation latency.

- On Figure 6.6(b), we remark that if we force the leading coefficient of the cube root polynomial approximant (which has degree 8) to be a power of 2 ($M' = 1$), the target latency falls to 14 cycles (instead of 16 cycles). However, so far, CGPE has only found programs in 15 cycles, and this new lower bound of 14 cycles has not been reached. In such a case, we do not know yet if the lower bound is to optimistic, or if the heuristics used in CGPE have indeed failed to achieve it.

Not only does the target latency speed up the whole generation process, but it also allows to conclude on the optimality of some evaluation programs. The following of this chapter presents a

heuristic we have implemented to converge quickly towards parenthesizations of low evaluation latency.

6.3 Optimized search of *best* parenthesizations

We have seen in the previous two sections how to build all the parenthesizations for evaluating a given bivariate polynomial (Section 6.1), and how to converge toward those having the minimal latency by restricting Algorithm 6.1 to schemes achieving a given target latency (Section 6.2). However, we know from numercial results of Section 6.1.6 that generating all the parenthesizations of minimal latency, even for "small" degrees, may be costly. For example, recall from Table 6.4 that getting the 10494 schemes of minimal latency in the degree-(6,1) case took about 2m30s. Hence, it turns out to be useful to implement heuristics, that reduce the combinatorics during the generation, and thus reduce at the same time the whole generation timing to, say, 1 second or less for the previous example.

This section presents an approach that is fully heuristic, in order to generate in a faster way efficient parenthesizations that reduce the latency of evaluation. A second constraint that we introduce here is the number of multiplications involved in each parenthesization generated. Hence at the end of the process, the generated parenthesizations are sorted according to (1) the latency of evaluation, and (2) the number of multiplications.

Let us first describe the heuristic, and then give some numerical results to illustrate its interest.

6.3.1 Recursive search of best splittings of polynomials

This approach is based on a recursive search of the best splitting of the polynomial to be evaluated. To understand the process, let us consider the following univariate polynomial $a(x)$, that we want to evaluate with a latency at most τ (computed either statically or dynamically using the techniques of Section 6.2):

$$a(x) = a_{0,0} \ +_1 \ a_{1,0} \cdot x \ +_2 \ \cdots +_{\mathrm{n}} \ a_{n,0} \cdot x^n \quad \text{with} \quad a_{n,0} \neq 0, \tag{6.22}$$

The heuristic we have implemented consists in finding a splitting of the polynomial $a(x)$ in (6.22), by proceeding at step i as follows.

- We consider that the addition $+_{\mathrm{i}}$ in (6.22) is the last operation of the parenthesization:

$$a(x) = \left(a_{0,0} \ +_1 \ a_{1,0} \cdot x \ +_2 \ \cdots a_{i-1,0} x^{i-1}\right) \ +_{\mathrm{i}} \ \left(a_{i,0} \ +_{\mathrm{i}} \ \cdots \ +_{\mathrm{n}} \ a_n \cdot x^n\right).$$

- And then, we search recursively efficient parenthesizations for both subpolynomials, on the left and on the right of this addition.

Since for evaluating this polynomial with a latency at most τ, both subpolynomials have to be evaluated with a latency at most $\tau' = \tau - A$ (with A the cost of the addition), at each level of recursion we reduce by A the target latency for both subpolynomials. Finally at each level of recursion $\neq 1$, the first step consists in computing the dynamic target latency for evaluating the considered subpolynomial, and if it is larger than the latency τ', that means that the subpolynomial cannot be evaluated in at most τ' cycles; and thus that splitting is discarded. For example, let us consider the degree-(8,1) bivariate polynomial $P(s,t) = 2^{-p-1} + s \cdot \sum_{i=0}^{8} a_i t^i$ used to implement the square root of FLIP, for example. Using the methods presented in Section 6.2, we determine that this

polynomial cannot be evaluated in less than 13 cycles on the ST231 processor. Consider now the first splitting:

$$\underbrace{\left(2^{-p-1} + s \cdot \sum_{i=0}^{7} a_i t^i\right)}_{P'(s,t)} + \underbrace{a_8 t^8}_{a'(t)}.$$

Using this heuristic, we will try to evaluate both subpolynomials $P'(s,t)$ and $a'(t)$ in 13-1=12 cycles. However, we know from Table 6.6 that evaluating $P'(s,t)$ still requires 13 cycles, even without delay in the indeterminate s. Hence, we can conclude that this splitting cannot succeed. This heuristic is finally illustrated in Figure 6.7 below.

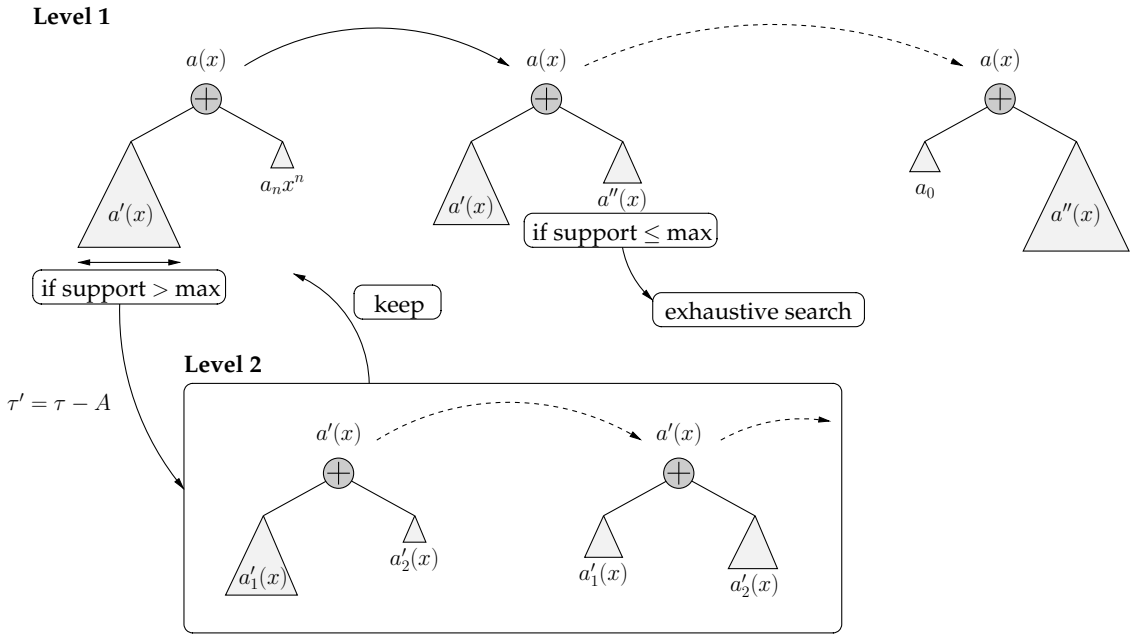


Figure 6.7: Optimized search of *best* parenthesizations of univariate polynomial.

To reduce the combinatorics, all along the process, we have introduced two other parameters.

- At a given step, if a subpolynomial has a support (number of coefficients) no larger than a "max" value, we launch the exhaustive algorithm on it, instead of the recursive-based method. In practice, when the support of the subpolynomial is no larger than 5, we are able to generate in a fast way all the parenthesizations of low latency.

- Since the number of schemes may still be extremely large, at each step of the process, we can keep only the best ones (in the sense of, first, latency of evaluation and, then, number of multiplications) among all those generated. In the remainder of the section, this parameter will be called "keep".

6.3.2 Numerical results

The questions that we can ask now are the following: (1) Using this heuristic, do we find any parenthesizations of lowest evaluation latency? (2) What is the impact of this heuristic on the timing of the whole generation process?

Let us first have a look at the generation of lower latency parenthesizations for evaluating our particular bivariate polynomials

$$P(s,t) = 2^{-p-1} + s\sum_{i=0}^{n_x} a_i t^i,$$

without any delay on the indeterminate s, that is, $D_s = 0$.

Degree n_x	5	6	7	8	9	10
Timing	$\leq$ 1s	$\leq$ 1s	1m56s	2m50s	2m26s	1s
Nb. of generated parenthesizations	155284	1221	10171239	14823887	16834164	43956
Latency	11	11	13	13	13	13
Nb. of multiplications	9	11	11	13	15	17

Table 6.7: First timings for the generation of parenthesizations for particular bivariate polynomials using our recursive search of best splittings.

From Table 6.7 we can observe that using this heuristic enables to converge in a very fast way towards parenthesizations of lower latency. For example, we have seen in Section 6.1.6, that generating the 447803 parenthesizations of minimal latency 11 for evaluating a degree-6 particular bivariate polynomial $2^{-1-p} + s\sum_{i=0}^{5} a_i t^i$ is done about 25s. Here, we generate 155284 schemes having this minimal latency, that is, about 34 %, in about one second, that is, 25 times faster.

What about the optimality of these parenthesizations? Using Tables 6.4 (for $n_x = 5$ and $n_x = 6$) and 6.6 (for $n_x = 7$, $n_x = 8$ and $n_x = 9$), we can conclude that the generated parenthesizations using this heuristic are optimal in terms of latency on unbounded parallelism.

Impact of the number of kept

Let us now consider the degrees $n_x = 7$, $n_x = 8$, and $n_x = 9$ in Table 6.7, which are the most costly to handle, and let us observe the impact of the number of parenthesizations kept at each step on the timing and the quality of the generated parenthesizations. Table 6.8 below displays the timing generation, the latency, and the number of multiplications of the generated schemes, for various values of "keep" parameter. (In Table 6.8 below, [13,13] indicates "of latency 13 cycles" and "involving 13 multiplications".) We observe on Table 6.8 below that, not suprisingly, if we

	Parameter "keep"					
	1	10	100	1000	10000	100000
$n_x = 7$	3s	3s	4s	5s	9s	45s
	[13,13]	[13,11]	[13,11]	[13,11]	[13,11]	[13,11]
$n_x = 8$	$\leq$1s	$\leq$1s	$\leq$ 1s	2s	7s	58s
	[13,13]	[13,13]	[13,13]	[13,13]	[13,13]	[13,13]
$n_x = 9$	$\leq$1s	$\leq$1s	$\leq$1s	2s	6s	1m03s
	[13,16]	[13,15]	[13,15]	[13,15]	[13,15]	[13,15]

Table 6.8: Impact of the value of the "keep" parameter on the timing of the generation and the "quality" ([latency, number of multiplications]) of the generated parenthesizations.

reduce the number of parenthesizations at each step, we reduce also the generation timing.

However, an interesting remark is the following: If we keep just one parenthesization at each step of the process, we may get a parenthesization with a few more multiplication(s) than the one we could have found otherwise. In Table 6.8, it is the case for $n_x = 7$ and $n_x = 9$.

From the results of Table 6.8, since most of the polynomials of FLIP are of degree $n_x = \{8, 9, 10\}$, for generating the efficient evaluation programs in a faster way, we will use this heuristic by keeping at each step only between 10 to 100 parenthesizations.

Interest of considering the delay during the generation

Let us now consider a real case of FLIP function implementation, and more particularly, the cube root $x^{1/3}$. The polynomial to be evaluated is $P(s,t) = 2^{-p-1} + s \cdot \sum_{i=0}^{9} a_i t^i$, with a delay on the indeterminate s of 9 cycles. Using this methodology, we have computed the timing for generating efficient schemes for this polynomial, whose results are displayed in Table 6.9 below.

	Parameter "keep"			
	1	10	100	1000
$n_x = 9$	34s	35s	34s	54s
	[16,15]	[16,14]	[16,14]	[16,14]

Table 6.9: Timing of the generation and quality ([latency, number of multiplications]) of the generated parenthesizations, for evaluating a degree-10 particular bivariate polynomial, $(n_x, n_y) = (9, 1)$.

In this case, we observe that as before that keeping just one parenthesization at each step may lead to an increase of the number of involved multiplications. But also, the timing is extremely more important than the one without delay on one indeterminate.

Actually, in this case, due to the delay of 9 cycles, the target latency (16 cycles) does not allow to reduce the combinatorics of our approach as much as expected, and the generation timing thus is significantly larger than when the delay is of 3 cycles (and the target latency of 13 cycles).

We could have chosen to generate schemes without taking the delay on s into account, and simply considered it after the generation. Therefore, we would have obtained in ≤ 1 second (see Table 6.8 for $n_x = 9$ and "keep= 1") a scheme with an evaluation latency of 13 cycles, as for example the following one:

```
((((c+(s*(a0+(t*a1))))+((s*(t*t))*(a2+(t*a3))))
  +((s*(t*t))*((t*t)*(a4+(t*a5)))))
 )+(((s*(t*t))*((t*t)*(t*t)))*((a6+(t*a7))+((t*t)*(a8+(t*a9))))))
```

The problem is that since s is known 9 cycles after t, the latency on unbounded paralelism as well as on a simplified model of the ST231 would have been in 19 cycles. (This latency of 19 cycles has been computed using CGPE and the scheduler presented in 6.4.2.)

That statement confirms the interest of taking the delay on the indeterminates into account during the generation, and not only after, in order to ensure the quality (in terms of latency of evaluation) of the generated programs.

6.4 Generating efficient and certified polynomial evaluation programs

This last section presents the general framework that we have implemented for generating *efficient* and *certified* polynomial evaluation programs for the implementation of mathematical

function. Recall that by *efficient*, we mean polynomial evaluation programs that reduce the evaluation latency (and the number of multiplications), and by *certified* we mean that we compute a bound on the evaluation error that is no larger than a target accuracy bound called η, as in Part I.

In [Mar08], a method is presented that transforms a given arithmetic expression in an equivalent one, so that the result expression is more accurate than the original one. This is in contrast with our approach here, where we want to be "accurate enough", but also as efficient as possible. In [CLM05] and [LV09], a methodology is proposed for implementing automatically mathematical functions in a given precision, and optimized for embedded processors. However this method is based on the evaluation of small degree polynomials, which are evaluated using Horner's rule.

CGPE (Code Generation for Polynomial Evaluation) has been developed during this thesis, and is actually the main part of this framework. It writes automatically efficient and certified codes, optimized for integer processors, for evaluating bivariate polynomialsin fixed-point arithmetic.

6.4.1 Presentation of the general framework

Here, the framework we propose aims at implementing automatically a given mathematical function F, defined as in Part I by

$$F(s,t) = 2^{-p-1} + s \cdot f(t), \quad \text{with} \quad s \in \mathcal{S} \quad \text{and} \quad t \in \mathcal{T}, \tag{6.23}$$

in a given precision p, and with a C program $\mathcal{P}$ entailing a final total error bounded by ϵ. Hence, from the description of the architecture and the description of the problem, we do the following:

- We determine the maximal error entailed by rounding the input s, called $\gamma(s)$, in order to make s representable on the considered architecture. Here, we assume that the input t is exactly representable, as in Chapters 3 and 4.
- We compute a polynomial approximant $a(t)$ with respect to the function $f(t)$ in (6.23) over $\mathcal{T}$ and a certified approximation error θ, such that the approximation error of $a(t)$ satisfies:

  $$\alpha(a) \leq \theta \quad \text{with} \quad \theta < \big(\epsilon - \gamma(s)\big) / \max\left(\mathcal{S}\right),$$

  where $\max(\mathcal{S})$ denotes $\max\big\{|x| : x \in \mathcal{S}\big\}$.
- Then, we determine an evaluation error bound η, as follows:

  $$\eta = \texttt{RoundDownward}(2^{-p-1} - \max(\mathcal{S}) \cdot \theta - \gamma(s)).$$

- Finally, we generate in an automatic way a C program $\mathcal{P}$ for evaluating $P(s,t) = 2^{-p-1} + s \cdot a(t)$ and whose evaluation error is strictly less than η.

This process is illustrated in Figure 6.8. The first two steps are done using Sollya [Che09], [Lau08], [CL], in a way similar to what is presented in Listings 3.1 and 4.6 for square root and division, in Chapters 3 and 4, respectively. By an iterative process, we can also generate a polynomial approximant having most of its coefficients forced to be powers of 2. Indeed, we start by considering each coefficient of the polynomial. Once we have tried to *structure* all the coefficients, we try to structure each pair of coefficients amoung those that can be structured. (Here, we say that a coefficient can be structured if once the new polynomial approximant is computed, its approximation error is still small enough. In particular, if a given $a_{i,j}$ cannot be structured, then no pair of coefficients containing $a_{i,j}$ can be structured neither.) We iterate like this until no n-uples of structured coefficients can be created.

Once the polynomial and the certified evaluation error bound are known, we compute using CGPE a set of efficient parenthesizations for evaluating this particular polynomial. It remains now to see how to select one efficient evaluation parenthesization or a set of efficient evaluation parenthesizations, so that we write the C code and the Gappa certificate.

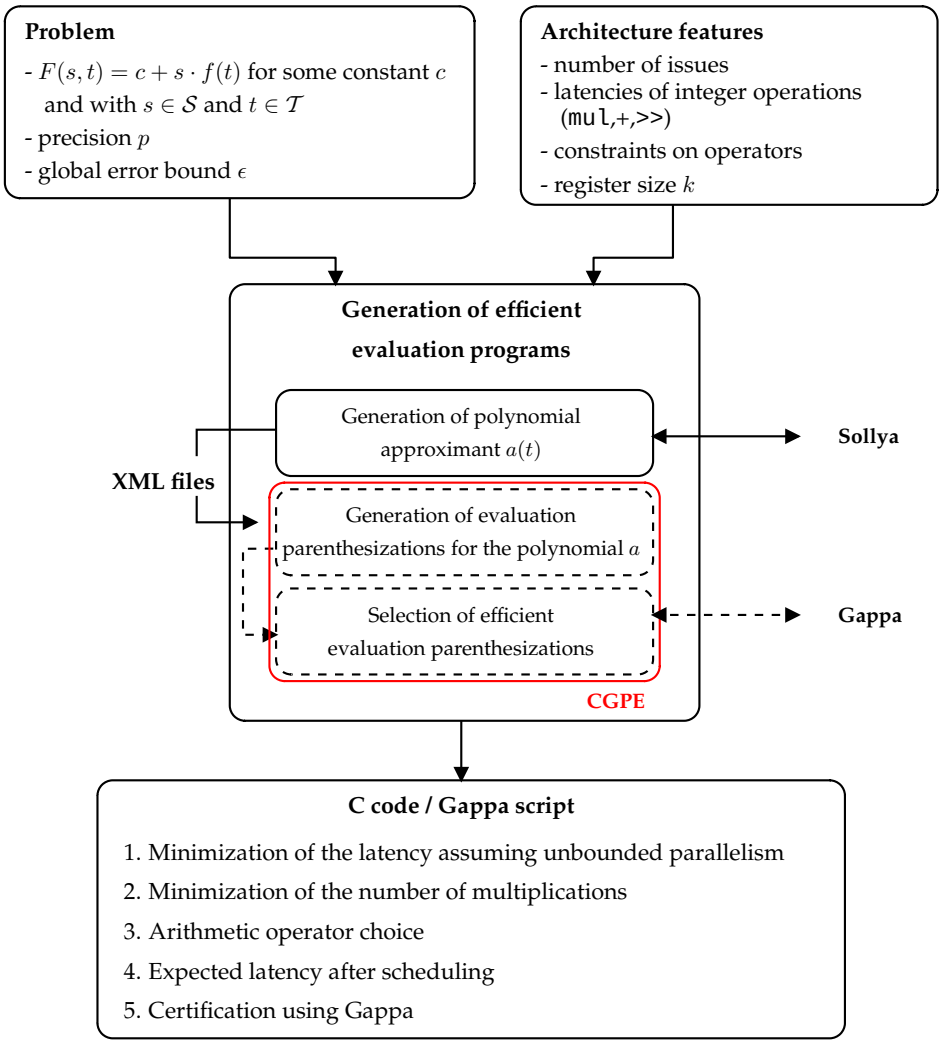


Figure 6.8: General framework for the automatic generation of efficient and certified polynomial evaluation programs.

6.4.2 Efficient parenthesization selection and certified code generation

At this stage, we have a set of efficient evaluation parenthesizations. We want now to check if one of them can be used effectively for the implementation of a function on a given target, typically the ST231. For a given parenthesization, our selection is done in three steps, called also *filters* since they enable to reduce at the end the number of efficient parenthesizations.

Arithmetic Operator Choice. We check if the given polynomial can be evaluated with this parenthesization using only unsigned integer arithmetic. This is done using certified interval arithmetic (MPFI [CLN+], see also [RR05]) as presented in Section 5.1.1, Chapter 5.

Scheduling on a simplified model of the ST231. If this parenthesization has passed the first filter, we check if it can be scheduled on the ST231 processor without any increase of latency.

Following for example what is done in [Ren08, p. 97,§4.4] we have implemented a scheduler based on classical list-scheduling, and which takes into account the constraints related to the bundling of the instructions on that architecture.

We could have scheduled this parenthesization directly on the target architecture (after having generated the corresponding C code). But our scheduler allows to take the structure of the problem into account, and more particularly the delay between the two indeterminates of our special bivariate polynomial.

C code and Gappa certificate generation. If the considered parenthesization passes also the second filter, it remains to check if the entailed evaluation error is strictly less than the required bound η. Hence, we finally generate the C code and the Gappa script corresponding to the considered parenthesization, and we check with Gappa [Mel], [Mel06] if the error entailed by the evaluation of the program is less than η. If it is not the case, the scheme is rejected and another one has to be considered.

This framework is heuristic and still at an experimental stage, and if no program is eventually found, we have to restart the process with other parameters. However, it has worked so far very well for all the functions we have implemented in FLIP and which have been detailed in Chapters 3 and 4.

6.4.3 Last examples of generated programs

First numerical results: generation of code for roots and their reciprocals

Our framework has been used for generating evaluation program for the function $x^{1/n}$ for $n \in \{-5,-4,-3,-2,-1,2,3,4,5\}$. Table 6.10 shows the timings for each step of the generation and filtering process. (The number between brackets indicates the numbers of parenthesizations.)

	$x^{1/2}$	$x^{-1/2}$	$x^{1/3}$	$x^{-1/3}$	$x^{1/4}$	$x^{-1/4}$	$x^{1/5}$	$x^{-1/5}$	x^{-1}
Parenthesization generation	172ms [50]	152ms [50]	53s [50]	56s [50]	169ms [50]	149ms [50]	53s [50]	53s [50]	168ms [50]
Arithmetic Operator Choice	6ms [31]	6ms [28]	7ms [32]	11ms [20]	5ms [31]	6ms [28]	7ms [32]	6ms [32]	4ms [8]
Scheduling	29s [2]	4m21s [1]	32ms [32]	132ms [17]	29s [2]	4m21s [1]	31ms [32]	32ms [32]	7s [5]
Certification (Gappa)	6s [2]	4s [1]	1m 38s [32]	1m07s [17]	6s [2]	4s [1]	1m38s [32]	1m37s [32]	11s [5]
Total time (≈)	35s	4m25s	2m31s	2m03s	35s	4m25s	2m31s	2m03s	18s

Table 6.10: Timings for certified code generation for roots and their reciprocals, using this general framework.

We can remark from Table 6.10 that the two most expensive steps are the scheduling on the simplified model of the ST231 architecture and the certification using Gappa. The impact of Gappa is discussed later. Here we focus on the impact of the scheduling. For example, for $n \in \{-2,-4\}$, we observe from Table 6.5 that these cases are the only ones where the latency on the simplified model of the ST231 (14 cycles) is 1 cycle more than the one on unbounded parallelism (13 cycles). That means that our scheduler tries to schedule in 13 cycles all the parenthesizations having passed the first filter. Since no parenthesization passes this first filter, it tries then to schedule all the

parenthesizations in 14 cycles. Hence, it does roughly two steps of scheduling. Consequently, the computation times are higher than in the cases where some parenthesizations can be scheduled with a latency equal to the target latency.

However, in Table 6.10 we observe the impact and the interest of the filters on the whole generation and filtering process. Hence, it shows the efficiency of the generation algorithms and heuristics, that we have presented in this thesis. Let us have a look at some others kinds of examples, before discussing some possible improvements.

Examples of functions not currently implemented in FLIP

Until now we have illustrated our methodologies and algorithms with functions which are implemented in the FLIP library, mainly nth roots and their reciprocals, as well as division and reciprocal. Let us finally consider two functions that are not currently implemented in the FLIP library:

$$f_1(t) = \log_2(1+t) \quad \text{with} \quad t \in [0.5, 1-2^{-23}] \qquad \text{and} \quad f_2(t) = 1/\sqrt{1+t^2} \quad \text{with} \quad t \in [0, 0.5-2^{-23}].$$

Using the framework presented in this section, we have generated some fast and certified C code for evaluating these functions on small intervals. Table 6.11 below summarizes the results of these generation steps.

	$f_1(t) = \log_2(1+x)$	$f_2(t) = 1/\sqrt{1+t^2}$
Polynomial approximant degree	(6,0)	(7,0)
Required approximation error	$\approx 2^{-24.99}$	$\approx 2^{-23.99}$
Approximation error bound θ	$\approx 2^{-27.72}$	$\approx 2^{-26.47}$
Evaluation error bound η	$\approx 2^{-25.23}$	$\approx 2^{-25.64}$
Parenthesization generation	15ms	11ms
	[50]	[50]
Arithmetic Operator Choice	2ms	4ms
	[2]	[12]
Scheduling	5ms	154ms
	[1]	[5]
Certification (Gappa)	921ms	5s
	[1]	[5]
Computed evaluation error (Gappa)	$\approx 2^{-28.84}$	$\approx 2^{-28.19}$
Total time	$\approx$ 1s	$\approx$ 5s

Table 6.11: Timings for the generation of $\log_2(1+t)$ and $1/\sqrt{1+t^2}$.

Finally, we can remark, not surprisingly, that the method succeeds for other functions than the ones presented all along this document. The interesting remark that we can do is that, on these last two examples, the most costly part of the generation is the certification with Gappa. Currently, we are implementing a certification phase using certified interval arithmetic with MPFI. This is still a preliminary study, but we already observe in practive that when the bound can be satisfied using MPFI, this certification is faster than the one using Gappa. Possible explanations may be due to the following reasons.

- Certifying with Gappa consists in an external call from CGPE to Gappa, while MPFI is compiled inside CGPE.

- MPFI computes a certified bound of the evaluation error, while Gappa checks if this evaluation error satisfies a given bound, which could be more expensive, especially when the evaluation error is close to this bound.
- Gappa uses interval arithmetic, but also rewriting rules and theorems, and consequently the certification with Gappa involves more calculations than the one with MPFI and thus may be slightly more costly.

Of course, notice that generally the results with Gappa are more accurate than the ones with MPFI. However, in our context, MPFI seems to be a good alternative for the certification of C codes.

Finally, from these results, improving the whole generation process would also rely on the improvement of the scheduling phase. We could use the scheduler of the ST231 compiler, which provides assembly code very close to the optimal (see Section 1.1), by compiling the corresponding C code. However this external call to the compiler has a cost, and we must be careful on the fact that this call has not to be more costly than a call to the list-scheduling algorithm.

Conclusion

The work presented in this document addresses the design and the implementation of an efficient software support for IEEE 754 floating-point arithmetic on integer processors, through a set of correctly-rounded mathematical operators, handling subnormal floating-point numbers, and handling special inputs. Currently, the most important challenge is the implementation of methodologies and tools dedicated to the automation of the implementation of mathematical operators, and not the implementation of these operators themselves. This thesis is dedicated to that purpose. More particularly, we have proposed a parametrized description for the implementation of some mathematical operators and a tool that generates efficient parenthesizations for the evaluation of bivariate polynomials. These two complementary study directions led to the two software developments, FLIP 1.0 and CGPE, and the two parts of this thesis.

Part I discusses the design and the implementation of *efficient* and *certified* software support for *binary32* floating-point arithmetic on embedded integer processors, especially optimized for the ST231 processor, which is a 4-issue VLIW integer processor of the ST200 processor family of STMicroelectronics. More particularly, one of the objectives of this part is to bring out various basic blocks from the implementation of correctly-rounded floating-point operators, and to present each of these basic blocks with a parametrized description and analysis. Hence, the implementation of an efficient and certified operator simply relies on the systematic use of these basic blocks. Moreover, this first part proposes a uniform approach for implementing roots and their reciprocals, and the division, which is based on the evaluation of a *single particular bivariate* polynomial. This general methodology has already been used for the implementation of several operators of the FLIP library, like addition and subtraction, multiplication, roots and their reciprocals, and division. This approach turns out to be very efficient, since it enables to achieve implementations up to about 1.95 faster than the ones of the previous version FLIP 0.3. This efficiency relies on the efficiency of the basic blocks and their systematic uses, on a better exploitation of the binary interchange format encoding, on proved rounding procedures, as well as on the optimized evaluation of the bivariate polynomial via a parenthesization automatically generated using CGPE. Notice first that each implementation leads to a certified C code for the *binary32* floating-point format. The interest of the certification is that we can thus control the errors (entailed by polynomial evaluation, for example) and then ensure correct rounding of the operator implementation. We have observed that the certification phase may be more or less difficult according to the function: for example, the certification of the polynomial evaluation code for the division is not as direct as for the other operators, and we had thus to implement a piecewise certification strategy. Notice also that here, for each operator, the implementation of the rounding procedure relies on a rounding condition, whose implementation is presented in this document according to the operator and consists in the inversion of the function to be implemented. It turns out that, for roots and their reciprocals, this inversion remains the last implementation part to be automated. This is a current work, which has already been started and illustrated for reciprocal square root [JR09a].

From now, we can remark that this inversion (for the implementation of roots and their recipro-

cals) may be costly. Hence let us consider faithful implementations (instead of correctly-rounded), which means that each implementation returns one of the two floating-point numbers closest to the exact result (and the exact result if it is a floating-point number). As shown in Table 6.12 for various functions, it enables to achieve implementations with a latency between 19 and 27 cycles,

Function	$x^{1/2}$	$x^{-1/2}$	$x^{1/3}$	$x^{-1/3}$	$x^{-1/4}$	x^{-1}	x/y
Timing (# cycles)	19	21	25	27	26	20	26

Table 6.12: Performances on ST231 processor, of faithful implementations.

that is, up to 1.6 faster than the corresponding correctly-rounded implementations. Such implementations may be a good alternative in audio and video domains, since these applications may not always require correct rounding.

Part II is then focused on the evaluation of polynomials in fixed-point arithmetic on integer processors like the ST231 processor. Indeed the efficiency of the implementations presented in Part I particularly relies on the optimized evaluation of the particular bivariate polynomial approximant. The difficulty of this part is due to the fact that, for a given polynomial, the number of parenthesizations may be extremely large, even for a "small" degree, and it remains long and tedious to find "by hand" an efficient parenthesization for this polynomial. By *efficient*, we mean a parenthesization that reduces the evaluation latency. Although we have shown that classical methods (Horner's rule, "second-order" Horner's rule, and Estrin-based methods) are accurate-enough for implementing various operators, they remain not really efficient, that is, not really well-adapted in terms of latency of evaluation for being used on integer processors, like the ST231. Moreover we have observed that the approaches based on coefficient adaptation (Knuth and Eve's algorithm and Paterson and Stockmeyer's algorithm, for example) seem to be ill-suited for this kind of architecture as well. That statement has motivated the implementation of algorithms and heuristics that automate the generation of efficient and certified C codes for the evaluation of polynomials. They have been integrated into CGPE, which has already enabled to generate the polynomial evaluation code used for several operator implementations of the FLIP library. Concerning this tool, we have observed that some improvements may be done through the speedup of the most costly parts, which are the scheduler on a simplified model of the ST231 processor and the certification using Gappa. Regarding the certification, we are currently studying the feasibility of certifying our C codes using certified interval arithmetic and MPFI, which seems to be faster and thus to be a good alternative to Gappa (but in this case, we lose the possibility of generating a Coq proof, as provided by Gappa).

About the certification of the implementations, let us emphasize that in this document, the certification is done at the C code level. Indeed the efficiency and the optimizations of the current compilers (like the ST231 compiler) may bring back the certification of the operators implemented. For example, let us consider the degree-3 polynomial $a(x) = \sum_{i=0}^{3} a_i \cdot x^i$ and the following parenthesization:

$$a_0 + \Big((a_1 \cdot x) + \big(\big((x \cdot x) \cdot a_2\big) + \big(x \cdot (x \cdot (a_3 \cdot x))\big)\big)\Big).$$

When compiling the corresponding C code on the ST231 processor, the generated assembly code implements the following parenthesization instead:

$$\big(a_0 + (a_1 \cdot x)\big) + \big(\big((x \cdot x) \cdot a_2\big) + \big(x \cdot (x \cdot (a_3 \cdot x))\big)\big),$$

which is slightly different. In fact, in practice, the second parenthesization is 1 cycle faster than the first one, and is favored by the compiler. However, the error entailed at runtime may differ from the one initially implemented in C and certified with Gappa. In this particular case, we can

then certify the assembly code in addition to the C code. However, the second certification phase may be costly and not well-adapted for being used in an automatic generation process, since each time we have first to compile the C code and then reread the generated assembly code to certify it. But also, the certification of the assembly code is architecture dependent and requires a good knowledge of the instruction set of each target for which we want to implement the operator.

To conclude on this work, the interest of CGPE comes from the fact that combined with the general method presented in Part I, it enables to generate in a fast way efficient and certified C codes for correctly-rounded (or at least faithful) implementations of mathematical operators.

A first extension to this work aims at study the efficiency of the approach presented in Part I for other standard formats. Throughout this document, we have presented parametrized algorithms for implementing mathematical operators, and we have illustrated these approaches with C codes for the *binary32* floating-point format. The interest of such an approach is that it enables to write quickly efficient code for various formats. Hence let us now consider the *binary64* floating-point format and, for example, the square root function. The implementation of this function relies on the evaluation of a degree-18 polynomial. Here the difficulties are due to the fact that the implementation is based on 64-bit arithmetic, which is simply emulated on the ST231 (for example, $64 \times 64 \rightarrow 64$ multiplications are implemented in software using several 32-bit operations). However, from now, we can already observe that this method enables to achieve a correctly-rounded square root in 171 cycles in RoundTiesToEven and for the *binary64* floating-point format, that is, with a speedup of 56.8 % compared to the *binary64* square root of STlib currently used by STMicroelectronics.

Finally, we have extended the techniques presented in Part I for the computation of sufficient error bounds and the implementation of correct rounding to the implementation of the square root function on FPGA's [dDJPR09], and compared to other methods usually used: iterative (SRT) and multiplicative (Newton-Raphson) methods. The implementation is done via the evaluation of several degree-2 univariate polynomials. On that context and for the *binary32* floating-point format, this approach seems to be an interesting alternative to multiplicative methods, since we gain in latency and slices to the detriment of an increase of the number of multipliers and blocks RAM used. More generally, a second extension of this work consists in studying the interest of all the techniques presented in this document for the implementation of correctly-rounded mathematical operators on other kinds of architectures like FPGA's.

Part III

Appendices

APPENDIX A

Notation

k	format width, such that $k = w + p$
p	precision, such that $p \geq 2$
$e_{\min}, e_{\max}$	extremal exponents, such that $e_{\min} = 1 - e_{\max}$, and $e_{\max} = 2^{w-1} - 1$
w	exponent width
b	exponent bias, such that $b = -e_{\min} + m_{x,0}$
α	smallest positive subnormal number, such that $\alpha = 2^{e_{\min}-p+1}$
Ω	largest finite positive floating-point number, such that $\Omega = (2 - 2^{1-p}) \cdot 2^{e_{\max}}$
x	finite binary floating-point number, such that $x = (-1)^{s_x} \cdot m_x \cdot 2^{e_x}$
$m_x = m_{x,0}.m_{x,1} \ldots m_{x,p-1}$	significand of floating-point number x, and its binary expansion: for $i \in \{0, \ldots, p-1\}$, $m_{x,i} \in \{0, 1\}$
λ_x	number of leading zeros in the binary expansion of m_x
e_x	exponent of floating-number x, such that $e_{\min} \leq e_x \leq e_{\max}$
s_x	sign of floating-point number x
$n_x = m_{x,0}$	"is normal" bit of x (that is. $n_x = 1$ if x is normal, and $n_x = 0$ if x is subnormal)
e'_x	scaled exponent of floating-number x, such that $e'_x = e_x - \lambda_x$
$m'_x = 1.m_{x,\lambda_x+1} \ldots m_{x,p-1}$	normalized significand of floating-point number x, with $m_{x,i} \in \{0, 1\}$ for $i\lambda_x + 1 \leq i < p$
X	k-bit unsigned integer giving the standard binary interchange encoding of x
M_x	k-bit unsigned integer encoding of m_x, such that $M_x = m_x \cdot 2^{k-1}$
E_x	w-bit unsigned integer encoding of the biased value of e_x, such that $E_x = e_x - e_{\min} + n_x$
M_{px}	k-bit unsigned integer encoding of m'_x, such that $M_{px} = m'_x \cdot 2^{k-1}$
D_x	w-bit unsigned integer, such that $D_x = E_x - n_x$

S_x	k-bit unsigned integer, such that $S_x = s_x \cdot 2^{k-1}$
$\circ$	rounding-direction attributes in precision p, $\circ \in \{\mathrm{RN}_p, \mathrm{RU}_p, \mathrm{RD}_p, \mathrm{RZ}_p\}$
$\mathbb{N}$	set of natural integers
$\mathbb{N}^*$	set of positive integers $\mathbb{N} \setminus \{0\}$
$\mathbb{Z}$	set of integers
$\mathbb{Z}^*_-$	set of negative integers $\mathbb{Z} \setminus \mathbb{N}$
$\mathbb{R}$	set of real numbers
$\mathbb{R}_+$	set of non negative real numbers
$\bar{\mathbb{R}}$	set of extended real numbers, $\bar{\mathbb{R}} = \mathbb{R} \cup \{-\infty, +\infty\}$
$[a, b]$	set of real numbers x such that $a \leq x \leq b$
$\{a, \ldots, b\}$	set of integers i such that $a \leq i \leq b$
$\alpha(a)$	approximation error of the polynomial a with respect to a function f over an interval $\mathcal{T}$
$\rho(\mathcal{P})$	evaluation error of the evalution program $\mathcal{P}$
θ	certified approximation error bound
η	certified evaluation error bound
$\mathrm{E}^{(k)}$	set of *valid* expressions of total degree exactly k
$\mathrm{P}^{(k)}$	set of expressions of the form $x^i y^j$, with $i, j \in \mathbb{N}$ and $i + j = k$

APPENDIX B

C code for implementing various basic integer operators

B.1 Implementation of $\max$ operators

```
static inline
int32_t max (int32_t a ,int32_t b)
{ return (a > b) ? a : b; }
```

```
static inline
uint32_t maxu (uint32_t a ,uint32_t b)
{ return (a > b) ? a : b; }
```

B.2 Implementation of $\min$ operators

```
static inline
int32_t min (int32_t a ,int32_t b)
{ return (a < b) ? a : b; }
```

```
static inline
uint32_t minu (uint32_t a ,uint32_t b)
{ return (a < b) ? a : b; }
```

B.3 Implementation of $32 \times 32 \rightarrow 32$ multiplications

```
static inline
uint32_t mul (uint32_t a ,uint32_t b)
{
  uint64_t t0 = a;
  uint64_t t1 = b;
  uint64_t t2 = (t0 * t1) >> 32;
  return t2;
}
```

```
static inline
int32_t mul64h (int32_t a ,int32_t b)
{
  int64_t t0 = a;
  int64_t t1 = b;
  int64_t t2 = (t0 * t1) >> 32;
  return t2;
}
```

B.4 Implementation of *count leading zeros* operator

```
static inline
uint32_t nlz(uint32_t x)
{
  uint32_t z = 0;
  if (x == 0) return(32);
  if (x <= 0x0000FFFF) {z = z + 16; x = x << 16;}
  if (x <= 0x00FFFFFF) {z = z +  8; x = x <<  8;}
  if (x <= 0x0FFFFFFF) {z = z +  4; x = x <<  4;}
  if (x <= 0x3FFFFFFF) {z = z +  2; x = x <<  2;}
  if (x <= 0x7FFFFFFF) {z = z +  1;}
  return z;
}
```

Bibliography

[AGS99] Ramesh C. Agarwal, Fred G. Gustavson, and Martin S. Schmookler. Series approximation methods for divide and square root in the Power3™ processor. In Israel Koren and Peter Kornerup, editors, *Proceedings of the 14th IEEE Symposium on Computer Arithmetic (ARITH'14)*, pages 116–123, Adelaide, Australia, 1999. IEEE Computer Society. [pages 4, 60, and 111]

[Ame85] American National Standards Institute and Institute of Electrical and Electronic Engineers. *IEEE Standard for Binary Floating-Point Arithmetic*. ANSI/IEEE Standard 754-1985, 1985. [pages 18 and 29]

[BBdD+04] Christian Bertin, Nicolas Brisebarre, Benoît Dupont de Dinechin, Claude-Pierre Jeannerod, Christophe Monat, Jean-Michel Muller, Saurabh-Kumar Raina, and Arnaud Tisserand. A Floating-point Library for Integer Processors. In *Proceedings of the 49th Annual Meeting International Symposium on Optical Science and Technology (SPIE'04)*, volume 5559, pages 101–111, Denver, CO, USA, 2004. SPIE. [page 3]

[BD04] Sylvie Boldo and Marc Daumas. A simple test qualifying the accuracy of Horner's rule for polynomials. *Numerical Algorithms*, 37(1-4):45–60, 2004. [page 147]

[Bol04] Sylvie Boldo. *Preuves formelles en arithmétiques à virgule flottante*. PhD thesis, École Normale Supérieure de Lyon, 46 allée d'Italie, F-69364 Lyon cedex 07, France, 2004. [page 147]

[Bru06] Christian Bruel. If-conversion SSA framework for partially predicated VLIW architectures. In *Digest of the 4th Workshop on Optimizations for DSP and Embedded Systems*, Manhattan, New York, NY, 2006. [page 12]

[BZ09] Richard Brent and Paul Zimmermann. *Modern Computer Arithmetic*. 2009. Version 0.3. Available at `http://www.loria.fr/~zimmerma/mca/mca-0.3.pdf`. [page 60]

[Che09] Sylvain Chevillard. *Évaluation efficace de fonctions numériques - Outils et exemples*. PhD thesis, Université de Lyon - École Normale Supérieure de Lyon, 46 allée d'Italie, F-69364 Lyon cedex 07, France, 2009. [pages 7, 71, 129, 162, and 190]

[CHN99] Marius Cornea-Hasegan and Bob Norin. IA-64 floating-point operations and the IEEE standard for binary floating-point arithmetic. *Intel Technology Journal*, 1999-Q4:1–16, 1999. [page 60]

[CHT02] Marius Cornea, John Harrison, and Ping Tak Peter Tang. *Scientific Computing on Itanium®-based Systems*. Intel Press, Hillsboro, OR, 2002. [pages 59, 60, 111, 146, and 166]

[CJL09] Sylvain Chevillard, Mioara Joldes, and Christoph Lauter. Certified and Fast Computation of Supremum Norms of Approximation Errors. In Javier D. Bruguera, Marius Cornea, Debjit DasSarma, and John Harrison, editors, *Proceedings of the 19th IEEE Symposium on Computer Arithmetic (ARITH'19)*, pages 169–176, Portland, OR, USA, 2009. IEEE Computer Society. [pages 75, 89, and 104]

[CK04] Martine Ceberio and Vladik Kreinovich. Greedy algorithms for optimizing multivariate Horner schemes. *SIGSAM Bulletin*, 38(1):8–15, 2004. [page 147]

[CL] Sylvain Chevillard and Christoph Lauter. Sollya. Available at `http://sollya.gforge.inria.fr/`. [pages 1, 7, 71, 129, 162, and 190]

[CL07] Sylvain Chevillard and Chritoph Lauter. A certified infinite norm for the implementation of elementary functions. In Aditya Mathur, W. Eric Wong, and Man Fai Lau, editors, *Proceedings of the 7th IEEE International Conference on Quality Software (QSIC'07)*, pages 153–160, Portland, OR, USA, 2007. IEEE Computer Society. [pages 75, 89, and 104]

[CLM05] Ray C. C. Cheung, Dong-U Lee, and Oskar Mencer. Automating custom-precision function evaluation for embedded processors. In Thomas M. Conte, Paolo Faraboschi, William H. Mangione-Smith, and Walid A. Najjar, editors, *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'05)*, pages 22–31, San Francisco, CA, USA, 2005. ACM. [page 190]

[CLN+] Sylvain Chevillard, Christoph Lauter, Hong Diep Nguyen, Nathalie Revol, and Fabrice Rouiller. Multiple Precision Floating-point Interval library. Available at `http://gforge.inria.fr/projects/mpfi/`. [pages 94, 150, and 191]

[CLR92] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press and McGraw-Hill Book Company, Sixth edition, 1992. [page 182]

[dD04] Benoît Dupont de Dinechin. From machine scheduling to VLIW instruction scheduling. *ST Journal of Research*, 1(2), 2004. Available at `http://cri.ensmp.fr/classement/2003`. [page 11]

[dDJPR09] Florent de Dinechin, Mioara Joldes, Bogdan Pasca, and Guillaume Revy. Racines carrées multiplicatives sur FPGA. In *Proceedings of RenPar'19, SympA'13, and CFSE'7*, Toulouse, France, 2009. [page 197]

[dDP] Florent de Dinechin and Bogdan Pasca. FloPoCo - Floating-Point Cores. Available at `http://www.ens-lyon.fr/LIP/Arenaire/Ware/FloPoCo/`. [page 1]

[DLDdD+] Catherine Daramy-Loirat, David Defour, Florent de Dinechin, Matthieu Gallet, Nicolas Gast, Christoph Lauter, and Jean-Michel Muller. CR-Libm, A library of correctly rounded elementary functions in double-precision (documentation). Available at `http://lipforge.ens-lyon.fr/www/crlibm/`. [pages 145 and 146]

[EIM+00] Miloš D. Ercegovac, Laurent Imbert, David W. Matula, Jean-Michel Muller, and Guoheng Wei. Improving Goldschmidt division, square root, and square root reciprocal. *IEEE Transactions on Computers*, 49(7):759–763, 2000. [page 59]

[EL94] Miloš D. Ercegovac and Tomas Lang. *Division and Square Root: Digit-Recurrence Algorithms and Implementations*. Kluwer Academic Publishers, Boston, 1994. [pages 3, 59, and 116]

[EL04] Miloš D. Ercegovac and Tomas Lang. *Digital Arithmetic*. Morgan Kaufmann Publishers, 2004. [pages 22, 26, 38, 42, 59, 69, and 70]

[Eve64] James Eve. The evaluation of polynomials. *Numerische Mathematik*, (6):17–21, 1964. [pages 78, 146, 161, and 162]

[FBF+00] Paolo Faraboschi, Geoffrey Brown, Joseph A. Fisher, Giuseppe Desoli, and Fred Homewood. Lx: a technology platform for customizable VLIW embedded processing. In *Proceedings of the 27th annual International Symposium on Computer Architecture (ISCA'00)*, pages 203–213. ACM Press, 2000. [page 10]

[FDL] Freely Distributable LIBM (fdlibm). Available at `http://www.netlib.org/fdlibm/`. [page 3]

[FFY05] Joseph A. Fisher, Paolo Faraboschi, and Cliff Young. *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann, 2005. [page 10]

[FHL+07] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding. *ACM Transactions on Mathematical Software*, 33(2), 2007. Available at `http://www.mpfr.org/`. [pages 4, 59, and 60]

[Fik67] Charles T. Fike. Methods of evaluating polynomial approximations in function evaluation routines. *Communications of the ACM*, 10(3):175–178, 1967. [page 161]

[GCC] Software floating point in GCC. Available at `http://gcc.gnu.org/wiki/Software_floating_point`. [page 3]

[GHH+01] Bruce Greer, John Harrison, Greg Henry, Wei Li, and Peter Tang. Scientific computing on the Itanium processor. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, Denver, CO, USA, 2001. [page 60]

[GKP94] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Longman Publishing Co., Inc., Second edition, 1994. [page 64]

[Gra] Torbjörn Granlund. GMP - The GNU Multiple Precision arithmetic library. Available at `http://gmplib.org/`. [page 133]

[Gre02] Robin Green. Faster Math Functions. *Tutorial at Game Developers Conference 2002*, 2002. Available at `http://www.research.scea.com/research/research.html`. [pages 6 and 166]

[Hau] John Hauser. The SoftFloat and TestFloat Packages. Available at `http://www.jhauser.us/arithmetic/`. [pages 3, 4, and 115]

[Hig02] Nick J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, Second edition, 2002. [page 147]

[HKST99] John Harrison, Ted Kubaska, Shane Story, and Ping Tak Peter Tang. The computation of transcendental functions on the IA-64 architecture. *Intel Technology Journal*, 1999-Q4, 1999. [pages 6 and 166]

[HP06] John L. Hennessy and David A. Patterson. *Computer Architecture – A quantitative approach*. Fourth edition, 2006. [pages 11 and 16]

[IEE08] IEEE Computer Society. *IEEE Standard for Floating-Point Arithmetic*. IEEE Standard 754-2008, 2008. [pages 1, 4, 5, 18, 20, 22, 29, 30, 36, 43, 49, 50, 60, 61, 63, 81, 84, 100, and 116]

[IM99] Cristina Iordache and David W. Matula. On infinitely precise rounding for division, square root, reciprocal and square root reciprocal. In Israel Koren and Peter Kornerup, editors, *Proceedings of the 14th IEEE Symposium on Computer Arithmetic (ARITH'14)*, pages 233–240, Adelaide, Australia, 1999. IEEE Computer Society. [page 65]

[Int99] International Organization for Standardization. *Programming Languages – C*. ISO/IEC Standard 9899:1999, Geneva, Switzerland, 1999. [pages 34, 54, 131, and 136]

[JKM+08] Claude-Pierre Jeannerod, Hervé Knochel, Christophe Monat, Guillaume Revy, and Gilles Villard. A new binary floating-point division algorithm and its software implementation on the ST231 processor. Technical Report RR2008-39, Laboratoire de l'Informatique du Parallélisme (LIP), 46 allée d'Italie, F-69364 Lyon cedex 07, France, 2008. [page 135]

[JKM+09] Claude-Pierre Jeannerod, Hervé Knochel, Christophe Monat, Guillaume Revy, and Gilles Villard. A new binary floating-point division algorithm and its software implementation on the ST231 processor. In Javier D. Bruguera, Marius Cornea, Debjit DasSarma, and John Harrison, editors, *Proceedings of the 19th IEEE Symposium on Computer Arithmetic (ARITH'19)*, pages 95–103, Portland, OR, USA, 2009. IEEE Computer Society. [pages 5, 60, 112, 134, 138, and 139]

[JKMR07] Claude-Pierre Jeannerod, Hervé Knochel, Christophe Monat, and Guillaume Revy. Faster floating-point square root for integer processors. In *Proceedings of the 2nd IEEE Symposium on Industrial Embedded Systems (SIES'07)*, Lisbon, Portugal, 2007. [page 60]

[JKMR08] Claude-Pierre Jeannerod, Hervé Knochel, Christophe Monat, and Guillaume Revy. Computing floating-point square roots via bivariate polynomial evaluation. Technical Report RR2008-38, Laboratoire de l'Informatique du Parallélisme (LIP), 46 allée d'Italie, F-69364 Lyon cedex 07, France, 2008. [pages 5, 60, 85, and 139]

[Jr.03] Henry S. Warren Jr. *Hacker's Delight*. Addison-Wesley, 2003. [page 79]

[JR09a] Claude-Pierre Jeannerod and Guillaume Revy. Optimizing correctly-rounded reciprocal square roots for embedded VLIW cores. In *Proceedings of the 43rd Asilomar Conference on Signals, Systems, and Computers (Asilomar'09)*, Pacific Grove, CA, USA, 2009. [pages 5, 60, and 195]

[JR09b] Claude-Pierre Jeannerod and Guillaume Revy. A uniform approach to software implementation of correctly-rounded roots and their reciprocals. 2009. In preparation. [page 80]

[Knu62] Donald E. Knuth. Evaluation of polynomials by computers. *Communications of the ACM*, 5(12):595–599, 1962. [pages 146 and 161]

[Knu98] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Third edition, 1998. [pages 78, 145, 146, 147, 153, 155, 161, 162, 180, and 181]

[Lau] Christoph Lauter. Metalibm. Available at http://lipforge.ens-lyon.fr/www/metalibm/. [page 1]

[Lau08] Christoph Lauter. *Arrondi correct de fonctions mathématiques - fonctions univariées et bivariées, certification et automatisation*. PhD thesis, Université de Lyon - École Normale Supérieure de Lyon, 46 allée d'Italie, F-69364 Lyon cedex 07, France, 2008. [pages 1, 7, 71, 129, 145, 146, 162, and 190]

[LV09] Dong-U Lee and John D. Villasenor. Optimized Custom Precision Function Evaluation for Embedded Processors. volume 58, pages 46–59, Washington, DC, USA, 2009. IEEE Computer Society. [page 190]

[Mar90] Peter W. Markstein. Computation of elementary functions on the IBM RISC System/6000 processor. *IBM Journal of Research and Development*, 34(1):111–119, 1990. [pages 16 and 60]

[Mar00] Peter W. Markstein. *IA-64 and Elementary Functions : Speed and Precision*. Hewlett-Packard Professional Books. Prentice Hall, 2000. [pages 3, 59, 60, and 111]

[Mar04] Peter W. Markstein. Software division and square root using Goldschmidt's algorithms. In *Proceedings of the 6th Conference on Real Numbers and Computers (RNC'6)*, pages 146–157, Dagstuhl, Germany, 2004. [pages 3 and 60]

[Mar08] Matthieu Martel. Enhancing the Implementation of Mathematical Formulas for Fixed-Point and Floating-point Arithmetics. In *Proceedings of the First International Workshop on Numerical Abstractions for Software Verification*, Princeton, NJ, USA, 2008. [page 190]

[MBCP07] Paolo Montuschi, Javier D. Bruguera, Luigi Ciminiera, and José-Alejandro Piñeiro. A Digit-by-Digit Algorithm for mth Root Extraction. *IEEE Transactions on Computers*, 56(12):1696–1706, 2007. [page 59]

[MBdD+09] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2009. [pages 19, 29, and 51]

[Mel] Guillaume Melquiond. Gappa - Génération automatique de preuves de propriétés arithmétiques. Available at http://lipforge.ens-lyon.fr/www/gappa/. [pages 8, 92, 129, 149, and 192]

[Mel06] Guillaume Melquiond. *De l'arithmétique d'intervalles à la certification de programmes*. PhD thesis, École Normale Supérieure de Lyon, 46 allée d'Italie, F-69364 Lyon cedex 07, France, 2006. [pages 8, 92, 129, 149, and 192]

[MM] David W. Matula and Lee D. McFearin. Extremal rounding test sets. Available at http://engr.smu.edu/~matula/extremal.html. [page 4]

[MM90] Paolo Montuschi and Marco Mezzalama. Survey of square rooting algorithms. *Computers and Digital Techniques, IEE Proceedings-E*, 137(1):31–40, 1990. [page 59]

[Mou09] Christophe Mouilleron. Personal Communication, 2009. [page 177]

[Mul06] Jean-Michel Muller. *Elementary functions: algorithms and implementation*. Birkhäuser, Second edition, 2006. [pages 29, 133, and 161]

[Obe99] Stuart F. Oberman. Floating point division and square root algorithms and implementation in the AMD-K7™ microprocessor. In Israel Koren and Peter Kornerup, editors, *Proceedings of the 14th IEEE Symposium on Computer Arithmetic (ARITH'14)*, pages 106–115, Adelaide, Australia, 1999. [pages 60 and 111]

[OF96] Stuart F. Oberman and Michael J. Flynn. Fast IEEE rounding for division by functional iteration. Technical Report CSL-TR-96-700, Computer Systems Laboratory, Dept. on Electrical Engineering and Computer Science, Stanford University, Stanford, CA, USA, 1996. [page 111]

[OF97a] Stuart F. Oberman and Michael J. Flynn. Design issues in division and other floating-point operations. *IEEE Transactions on Computers*, 46(2):154–161, 1997. [page 111]

[OF97b] Stuart F. Oberman and Michael J. Flynn. Division algorithms and implementations. *IEEE Transactions on Computers*, 46(8):833–854, 1997. [pages 3 and 111]

[Pan66] Victor Y. Pan. On Methods of Computing the Values of Polynomials. *Uspekhi Matematicheskikh Nauk*, pages 103–134, 1966. [page 146]

[PB02] José-Alejandro Piñeiro and Javier D. Bruguera. High-speed double-precision computation of reciprocal, division, square root and inverse square root. *IEEE Transactions on Computers*, 51(12):1377–1388, 2002. [pages 3, 59, 60, and 111]

[PBLM08] José-Alejandro Piñeiro, Javier D. Bruguera, Fabrizio Lamberti, and Paolo Montuschi. A radix-2 digit-by-digit architecture for cube root. *IEEE Transactions on Computers*, 57(4):562–566, 2008. [page 59]

[PPB03] Daniel Piso, José-Alejandro Piñeiro, and Javier. D. Bruguera. Analysis of the impact of different methods for division/square root computation in the performance of a superscalar microprocessor. *Journal of Systems Architecture*, 49(12-15):543–555, 2003. [page 59]

[PS73] Mike S. Paterson and Larry J. Stockmeyer. On the number of nonscalar multiplications necessary to evalutate polynomials. *SIAM Journal on Computing*, 2(1):60–66, 1973. [pages 78, 146, and 163]

[PS00] Juan Manuel Peña and Thomas Sauer. On the multivariate Horner scheme. *SIAM Journal on Numerical Analysis*, 37(4):1186–1197, 2000. [page 147]

[PT09] Ricardo Pachón and Lloyd N. Trefethen. Barycentric-Remez algorithms for best polynomial approximation in the chebfun system. *BIT Numerical Mathematics*, 49(4):721–741, 2009. [page 71]

[Rai06] Saurabh-Kumar Raina. *FLIP: a Floating-point Library for Integer Processors*. PhD thesis, École Normale Supérieure de Lyon, 46 allée d'Italie, F-69364 Lyon cedex 07, France, 2006. [pages 3, 16, 60, 111, and 115]

[Rem34] Evgeny Y. Remez. Sur le calcul effectif des polynômes d'approximation de Tchebichef. *Comptes rendus hebdomadaires des séances de l'Académie des Sciences*, 199:337–340, 1934. [pages 71 and 88]

[Ren08] Qian Ren. *Optimizing Behavioral Transformations using Taylor Expansion Diagrams*. PhD thesis, University of Massacusetts Amherst, 2008. [page 192]

[Rev] Guillaume Revy. CGPE - Code Generation for Polynomial Evaluation. Available at `http://cgpe.gforge.inria.fr/`. [pages 1, 129, and 177]

[Rev06] Guillaume Revy. Analyse et implantation d'algorithmes rapides pour l'évaluation polynomiale sur les nombres flottants. Master's thesis, École Normale Supérieure de Lyon, 46 allée d'Italie, F-69364 Lyon cedex 07, France, 2006. [pages 78, 155, 161, 162, and 163]

[RR05] Nathalie Revol and Fabrice Rouillier. Motivations for an arbitrary precision interval arithmetic and the MPFI library. *Reliable Computing*, 11(4):275–290, 2005. [pages 94, 150, and 191]

[SM06] Nathan Sidwell and Joseph Myers. Improving Software Floating Point Support. In *Proceedings of GCC Developer's Summit 2006*, pages 211–218, Ottawa, Canada, 2006. [page 3]

[ST208a] *ST200 VLIW Series – ST200 Micro Toolset compiler manual*, 2008. [page 11]

[ST208b] *ST231 core and instruction set architecture – Reference manual*, 2008. [pages 12, 13, 15, 16, 116, 122, and 148]

[Ste98] Gilbert W. Stewart. *Afternotes goes to Graduate School - Lectures on Advanced Numerical Analysis*. Society for Industrial and Applied Mathematics (SIAM), 1998. [page 71]

[SW99] Michael J. Schulte and Kent E. Wires. High-speed inverse square roots. In Israel Koren and Peter Kornerup, editors, *Proceedings of the* 14*th IEEE Symposium on Computer Arithmetic (ARITH'14)*, pages 124–131, Adelaide, Australia, 1999. IEEE computer society. [page 60]

[Tan05] Andrew Tanenbaum. *Architecture de l'Ordinateur*. Fifth edition, 2005. [pages 11 and 16]

[Ylö06] Jyri Ylöstalo. Function approximation using polynomials. *IEEE Signal Processing Magazine*, 23(5):99–102, 2006. [page 145]

[Zim08] Paul Zimmermann. Implementation of the reciprocal square root in MPFR. In Annie Cuyt, Walter Krämer, Wolfram Luther, and Peter Markstein, editors, *Numerical Validation in Current Hardware Architectures*, number 08021 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2008. [page 60]

Abstract

Today some embedded systems still do not integrate their own floating-point unit, for area, cost, or energy consumption constraints. However, this kind of architectures is widely used in application domains highly demanding on floating-point calculations (multimedia, audio and video, or telecommunications). To compensate this lack of floating-point hardware, floating-point arithmetic has to be emulated efficiently through a software implementation.

This thesis addresses the design and implementation of an efficient software support for IEEE 754 floating-point arithmetic on embedded integer processors. More specifically, it proposes new algorithms and tools for the efficient generation of fast and certified programs, allowing in particular to obtain C codes of very low latency for polynomial evaluation in fixed-point arithmetic. Compared to fully hand-written implementations, these tools allow to significantly reduce the development time of floating-point operators.

The first part of the thesis deals with the design of optimized algorithms for some binary floating-point operators, and gives details on their software implementation for the *binary32* floating-point format and for some embedded VLIW integer processors like those of the STMicroelectronics ST200 family. In particular, we propose here a uniform approach for correctly-rounded roots and their reciprocals, and an extension to division. Our approach, which relies on the evaluation of a single bivariate polynomial, allows higher ILP-exposure than previous methods and turns out to be particularly efficient in practice. This work allowed us to produce a fully revised version of the FLIP library, leading to significant gains compared to the previous version.

The second part of the thesis presents a methodology for automatically and efficiently generating fast and certified C codes for the evaluation of bivariate polynomials in fixed-point arithmetic. In particular, it consists of some heuristics for computing highly parallel, low-latency evaluation schemes, as well as some techniques to check if those schemes remain efficient on a real target, and accurate enough to ensure correct rounding of the underlying operator implementations. This approach has been implemented in the software tool CGPE (Code Generation for Polynomial Evaluation). We have used our tool to quickly generate and certify significant parts of the codes of FLIP.

Keywords: floating-point arithmetic, fixed-point arithmetic, polynomial evaluation, code generation and certification, embedded integer processor.

Résumé

Aujourd'hui encore, certains systèmes embarqués n'intègrent pas leur propre unité flottante, pour des contraintes de surface, de coût et de consommation d'énergie. Cependant, ce type d'architecture est largement utilisé dans des domaines d'application extrêmement exigeants en calculs flottants (le multimédia, l'audio et la vidéo ou les télécommunications). Pour compenser le fait que l'arithmétique flottante ne soit pas implantée en matériel, elle doit être émulée efficacement à travers une implantation logicielle.

Cette thèse traite de la conception et de l'implantation d'un support logiciel efficace pour l'arithmétique virgule flottante IEEE 754 aux processeurs entiers embarqués. Plus spécialement, elle propose de nouveaux algorithmes et outils pour la génération efficace de programmes à la fois rapides et certifiés, permettant notamment d'obtenir des codes C de très faibles latences pour l'évaluation polynomiale en arithmétique virgule fixe. Comparés aux implantations complètement écrites à la main, ces outils permettent de réduire de manière significative le temps de développement d'opérateurs flottants.

La première partie de la thèse traite de la conception d'algorithmes optimisés pour certains opérateurs flottants en base 2, et donne des détails sur leur implantation logicielle pour le format virgule flottante *binary32* et pour certains processeurs VLIW entiers embarqués comme ceux de la famille ST200 de STMicroelectronics. En particulier, nous proposons ici une approche uniforme pour l'implantation correctement arrondie des racines et de leur inverse, ainsi qu'une extension à la division. Notre approche, qui repose sur l'évaluation d'un seul polynôme bivarié, permet d'exprimer un plus haut degré de parallélisme d'instruction (ILP) que les méthodes précédentes, et s'avère particulièrement efficace en pratique. Ces travaux nous ont permis de fournir une version complètement remaniée de la bibliothèque FLIP, entraînant des gains significatifs par rapport à la version précédente.

La deuxième partie de la thèse présente une méthodologie pour générer automatiquement et efficacement des codes C rapides et certifiés pour l'évaluation de polynômes bivariés en arithmétique virgule fixe. En particulier, elle consiste en un ensemble d'heuristiques pour calculer des schémas d'évaluation très parallèles et de faible latence, ainsi qu'un ensemble de techniques pour vérifier si ces schémas restent efficaces sur une architecture cible réelle et suffisamment précis pour garantir l'arrondi correct de l'implantation des opérateurs sous-jacente. Cette approche a été implantée dans l'environnement logiciel CGPE (Code Generation for Polynomial Evaluation). Nous avons ainsi utilisé notre outil pour générer et certifier rapidement des parties significatives des codes de la bibliothèque FLIP.

Mots-clés : arithmétique virgule flottante, arithmétique virgule fixe, évaluation polynomiale, génération et certification de code, processeur entier embarqué.