Custom floating-point arithmetic for integer processors: algorithms, implementation, and selection

Jingyan Lu Jourdan

N ° d'ordre: 762
N ° attribué par la bibliothèque: 2012ENSL0762

ÉCOLE NORMALE SUPÉRIEURE DE LYON - UNIVERSITÉ DE LYON

Laboratoire de l'Informatique du Parallélisme

THÈSE

présentée et soutenue publiquement le 15 novembre 2012 par

Jingyan JOURDAN-LU

pour l'obtention du grade de

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

spécialité: Informatique

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

Custom floating-point arithmetic for integer processors: algorithms, implementation, and selection

Directeur de thèse :	Jean-Michel MULLER
Co-directeurs de thèse :	Claude-Pierre JEANNEROD
	Christophe MONAT
Après avis des rapporteurs :	Elisardo ANTELO
	Sid TOUATI

Devant la commission d'examen formée de :

Pr. Elisardo ANTELO	Universidade de Santiago de Compostela
M. Claude-Pierre JEANNEROD	INRIA & ENS Lyon
Pr. Daniel MÉNARD	INSA Rennes
M. Christophe MONAT	STMicroelectronics Grenoble
M. Jean-Michel MULLER	CNRS & ENS Lyon
M. Erven ROHOU	INRIA Rennes
Pr. Sid TOUATI	Université de Nice Sophia Antipolis

Abstract

Media processing applications typically involve numerical blocks that exhibit regular floating-point computation patterns. For processors whose architecture supports only integer arithmetic, these patterns can be profitably turned into custom operators, coming in addition to the five basic ones (+, −, ×, / and $\sqrt{}$), but achieving better performance by treating more operations.

This thesis addresses the design of such custom operators as well as the techniques developed in the compiler to select them in application codes.

We have designed optimized implementations for a set of custom operators which includes squaring, scaling, adding two nonnegative terms, fused multiply-add, fused square-add ($x^2 + z$ with $z \geqslant 0$), two-dimensional dot products (DP2), sums of two squares, as well as simultaneous addition/subtraction and sine/cosine. With novel algorithms targeting high instruction-level parallelism and detailed here for squaring, scaling, DP2, and sin/cos, we achieve speedups of up to 4.2x for individual custom operators even when subnormal numbers are fully supported.

Furthermore, we introduce the optimizations developed in the ST231 C/C++ compiler for selecting such operators. Most of the selections are achieved at high level, using syntactic criteria. However, for fused square-add, we also enhance the framework of integer range analysis to support floating-point variables in order to prove the required positivity condition $z \geqslant 0$.

Finally, we provide quantitative evidence of the benefits to support this selection of custom operations: on DSP kernels and benchmarks, our approach allows us to be up to 1.59x faster compared to the sole usage of basic ones.

Keywords: IEEE floating-point arithmetic; custom operator; embedded integer processor; VLIW architecture; compiler optimization; compiler code selection.

Résumé

Les applications multimédia se composent généralement de blocs numériques exhibant des schémas de calcul flottant réguliers. Sur les processeurs sans support architectural pour l'arithmétique flottante, ils peuvent être profitablement transformés en opérateurs dédiés, s'ajoutant aux cinq opérateurs élémentaires (+, −, ×, / et $\sqrt{}$) : en traitant plus d'opérations simultanément, ils permettent d'obtenir de meilleures performances.

Cette thèse porte sur la conception de tels opérateurs, et les techniques de compilation mises en oeuvre pour les sélectionner.

Nous avons réalisé des implémentations optimisées pour un ensemble d'opérateurs dédiés : élévation au carré, mise à l'échelle, fused multiply-add et sa spécialisation $x^2 + z$ avec $z \geqslant 0$, produit scalaire en dimension deux (DP2) et sa spécialisation comme somme de deux carrés, et enfin addition/soustraction simultanées et sinus/-cosinus simultanés. En proposant de nouveaux algorithmes cherchant à maximiser le parallélisme d'instructions et détaillés ici pour l'élévation au carré, la mise à l'échelle, DP2 et sinus/cosinus, nous obtenons des accélérations d'un facteur allant jusqu'à 4.2 par appel, et ce même en présence de nombres dénormaux.

Nous détaillons également les changements apportés dans le compilateur pour effectuer la sélection. La plupart des opérateurs sont sélectionnés au niveau syntaxique. Cependant, pour certains opérateurs, nous avons dû améliorer l'analyse d'intervalles entiers pour prendre en compte les variables de type flottant, afin de prouver certaines conditions de positivité requises à leur sélection.

Enfin, nous apportons la preuve en pratique de la pertinence de cette approche : sur des noyaux typiques du traitement du signal et sur certaines applications, nous mesurons une amélioration de performance allant jusqu'à 1.59x en comparaison avec la performance obtenue avec les seuls opérateurs élémentaires.

Mots-clés : IEEE arithmétique virgule flottante; opérateur dédié; processeur embarqué entier; architecture VLIW; optimisation du compilateur; sélection du code par le compilateur.

Table of Contents

Appendices

Notation

p	precision in bits, such that $p \geqslant 2$
w	exponent field width in bits
k	storage width in bits, such that $k = w + p$
$e_{\min}, e_{\max}$	extremal exponents, such that $e_{\min} = 1 - e_{\max}$, and $e_{\max} = 2^{w-1} - 1$
α	smallest positive binary subnormal number, such that $\alpha = 2^{e_{\min}-p+1}$
Ω	largest finite binary floating-point number, such that $\Omega = (2 - 2^{1-p}) \cdot 2^{e_{\max}}$
x	finite binary floating-point datum, such that $x = (-1)^{s_x} \cdot m_x \cdot 2^{e_x}$
s_x	sign bit of x
e_x	exponent of x, such that $e_{\min} \leqslant e_x \leqslant e_{\max}$
$m_x = (m_{x,0}.m_{x,1}m_{x,2}...m_{x,p-1})_2$	significand of x and its binary expansion: $m_x = \sum_{i=0}^{p-1} m_{x,i} \cdot 2^{-i}$ with $m_{x,i} \in \{0,1\}$ for $i \in \{0, \ldots, p-1\}$
λ_x	number of leading zeros in the binary expansion of m_x
X	k-bit unsigned integer giving the standard binary encoding of x
S_x	k-bit unsigned integer, such that $S_x = s_x \cdot 2^{k-1}$
E_x	w-bit unsigned integer encoding the biased value of e_x, such that $E_x = e_x - e_{\min} + m_{x,0}$
D_x	w-bit unsigned integer, such that $D_x = e_x - e_{\min}$
M_x	k-bit unsigned integer giving integral significand, such that $M_x = m_x \cdot 2^{p-1}$
$\lvert X \rvert$	encoding of $\lvert x \rvert$, such that $\lvert X \rvert = X \bmod 2^{k-1}$

$Q_{i.f}$	a $Q_{i.f}$ number is a rational number of the form $A/2^f$ with A an unsigned integer having $i+f$ bits
$\circ$	rounding-direction attributes, such that $\circ \in \{\mathrm{RN}, \mathrm{RU}, \mathrm{RD}, \mathrm{RZ}\}$
$\iota(x)$	an IEEE standard integer encoding of floating-point datum x
$\lfloor x \rfloor$	largest integer less than or equal to x
$\lceil x \rceil$	smallest integer greater than or equal to x
$\{x\}$	fractional value of x, such that $\{x\} = x - \lfloor x \rfloor$
$a \bmod b$	for $a, b \in \mathbb{N}$ with $b \neq 0$, the remainder $a - \lfloor a/b \rfloor b$ in the Euclidean division of a by b
$[a, b]$	for $a, b \in \mathbb{R}$, the set of all real numbers x such that $a \leqslant x \leqslant b$
$[a, b)$	for $a, b \in \mathbb{R}$, the set of all real numbers x such that $a \leqslant x < b$
$(a, b]$	for $a, b \in \mathbb{R}$, the set of all real numbers x such that $a < x \leqslant b$
(a, b)	for $a, b \in \mathbb{R}$, the set of all real numbers x such that $a < x < b$
$[\mathcal{S}]$	for a true-or-false statement $\mathcal{S}$, $[\mathcal{S}] = 1$ if $\mathcal{S}$ is true, $[\mathcal{S}] = 0$ if $\mathcal{S}$ is false
$\mathbb{R}$	set of real numbers
$\mathbb{Z}$	set of integers
$\mathbb{N}$	set of natural numbers
$\mathbb{F}$	set of binary$k$ finite floating-point numbers
$\wedge, \vee, \oplus, \neg$	for any two integers $i, j \in \{0, 1\}$, $i \wedge j$, $i \vee j$, $i \oplus j$, $\neg i$ denote, respectively, the integers ij, $i + j - ij$, $i + j \bmod 2$, $1 - i$
$I_{<x}$	for a real interval I and a real number x, $I_{<x}$ denotes the set of real numbers y such that $y \in I$ and $y < x$
$I_{\geqslant x}$	for a real interval I and a real number x, $I_{\geqslant x}$ denotes the set of real numbers y such that $y \in I$ and $y \geqslant x$

List of Tables

List of Figures

Acknowledgments

It is a great gift of life to work with happiness. I appreciate the chance that life has bestowed on me to work on interesting topics and to work with nice colleagues in the past few years.

Nothing can be achieved easily. Even when you enjoy the fun of work, there are always difficult moments. I would like to give my gratitude to the people who have supported me during my research project. Without you, this work would not have been possible.

My deepest gratitude goes to my thesis supervisors: Claude-Pierre Jeannerod, Christophe Monat, and Jean-Michel Muller, for making this research possible. Claude-Pierre has spent a lot of time, effort and dedication into this research. Thank you. I am sincerely grateful to Christophe for his patience, enthusiasm, and kind concern regarding my academic requirements. I thank Jean-Michel for his sincerity and encouragement, which I will never forget. All their support, guidance, advice throughout the project, have abundantly helped me to overcome all the obstacles in the completion of this research work.

My deepest gratitude is also due to the other members of the PhD committee: Elisardo Antelo, Daniel Ménard, Erven Rohou, and Sid Touati. I thank them for carefully reading my thesis and offering opening perspectives. Before the defense, Professors Antelo and Touati have done a great job as reviewers and provided me with constructive and valuable feedback, which has helped me a lot to improve the manuscript. Their sharp and clear ideas have been extremely valuable.

I owe special thanks to Christian Bertin, the head of the Compilation Expertise Center at STMicroelectronics, and Florent de Dinechin, the head of the Computer Arithmetic group at ENS de Lyon, for all their support, and especially their arrangements for funding my research project.

Special thanks also go to Nathalie Revol, Stephen Clarke, and Hervé Knochel. Nathalie has given me important advice on several presentations for international conferences as well as for my PhD defense. Stephen and Hervé have helped me with some very important technical details during the developments into the compiler. Thank you.

I would also like to express my gratitude to colleagues at the Computer Arithmetic group and at STMicroelectronics for their invaluable assistance. I thank Christoph Lauter, Sylvain Chevillard, Mioara Joldes, Guillaume Melquiond, Guillaume Revy, and Christophe Mouilleron for their innovative work on the software tools for computer arithmetic. I also thank all the members of CEC, who have developed such a robust compiler where I was able to integrate easily optimizations for floating-point applications. It is really an honor for me to work with you all.

Last but not the least, I would like to express my love and gratitude to my beloved family for their understanding and endless love. I love you.

Chapter 1

Introduction

1.1 Context and motivations

Even though media processing applications may rely intensively on floating-point computations, some modern embedded media processors such as the ST231 from the STMicroelectronics ST200 VLIW family do not contain floating-point hardware and provide architectural support only for integer arithmetic. This choice avoids paying high costs on silicon surface and power consumption.

Yet, this trade-off has to be compensated: a first approach would be to convert the applications to some fixed-point [CC99, AC00, MCCS02] or block floating-point format [MCS05]. With such techniques, ensuring accuracy can however be fairly complex, and sometimes costly and unsustainable. A second approach, which is the one we favor here, consists in designing a high-performance floating-point support.

The design and implementation of such a software library is critical in several aspects: not only its performance must enable key applications to reach an acceptable performance level, but the compliance with the IEEE 754-2008 standard [IEE08] must not be compromised.

Starting point. In order to achieve good-enough performance without sacrificing for accuracy, a first step is to optimize the five basic arithmetic operations by taking into account some features of the target architecture (parallelism, large multipliers, leading-zero counters,...). This was achieved by FLIP 1.0 (Floating-point Library for Integer Processors) [JR09a, Rev09], with new algorithms exposing high instruction-level parallelism (ILP). That library thus better exploits the VLIW architecture of the ST231 processor than the reference library SoftFloat [Hau] and in practice, the latency of each operator on ST231 was reduced by a factor of 1.85 to 5.21, as the table below shows:

	$+$	$-$	$\times$	$/$	$\sqrt{\ }$
SoftFloat	48	49	31	177	95
FLIP 1.0	26	26	21	34	23
speedup	1.85	1.88	1.48	5.21	4.13

Such speedups (which are given here assuming single precision, rounding 'to nearest even', and subnormal support) have been achieved by a combination of techniques, among which an optimized use of the IEEE 754 format encodings, a novel algorithmic approach based on highly-parallel polynomial evaluation for computing accurate approximations of functions like division and square root, and also some compiler optimizations; interestingly enough and unlike what is sometimes believed, the overhead for supporting subnormals (that is, the tiny floating-point numbers allowing gradual underflow) turned out to be extremely reasonable (for example, 5 cycles out of 34 for division and 2 out of 23 for square root) [Rev09, BJJL$^+$10].

Going beyond basic operators with custom operators. Embedded processing application codes and benchmarks typically involve numerical blocks that exhibit particular patterns, which we may refer to as being *non-generic* or *custom*. For example, Euclidean norm calculations consist of square rooting a sum of squares; also, radix-2 FFTs and arithmetic over the complex numbers use two-dimensional dot products (DP2). Thus, a second step to increase further the performances of such applications on integer processors like the ST231 is to design optimized custom operators like square and DP2, that will then be added to the existing basic arithmetic software support and selected at compile time. In fact, we can group custom operators into three categories, defined as follows:

- A *fused operator* replaces a set of two or more floating-point operators by a single one. Examples include the fused multiply-add (FMA) operation $xy + z$, as well as DP2 mentioned above.
- A *specialized operator* replaces a generic operator when the compiler can prove properties about its arguments. A typical example is that of square replacing a generic product xy whenever x equals y; another example is that of fused square-add (FSA) $x^2 + z$ with $z \geqslant 0$, which replaces FMA whenever x equals y and z is known to be nonnegative.
- A *paired operator* simultaneously evaluates two operators on the same input. For example, given floating-point input x and y, the so-called addsub operator will compute the pair $(x + y, x - y)$.

Note that some fused operators may in fact be fully specified by a standard like IEEE 754-2008. For example, this is the case of FMA and of functions like reciprocal square root and hypotenuse which compute, respectively, $1/\sqrt{x}$ and $\sqrt{x^2 + y^2}$ with just one rounding error; see [IEE08, Table 9.1].

In hardware, some custom operators have been studied extensively. For example, we refer to [SEES08, Sal09, SS12] and the references therein for DP2 and addsub designs applied to FFT butterfly units. (Note that in these works addsub is called 'fused' rather than 'pair'.) Another example is the computation of qth roots, for which algorithms and architectures have been proposed in [VB11]. In addition, customization is now common practice for FPGAs due to the high flexibility offered by such architectures [dDP11] and for application-specific instruction-set processors (ASIPs), methodologies on automating custom floating-point units are proposed [CP07, CP09].

In software, several custom operators have been considered for processors having floating-point hardware capabilities [CHT02, Mar03]. For integer processors like the ST231, division has been specialized to reciprocal [Rev09, p. 4] and examples of fused operators include reciprocal square, third, and fourth roots [Rev09, JR09b]. Such operators are however not always critical ones in DSP applications, where most of the computation time is spent on FMA-like patterns, that is, expressions made of additions and various kinds of multiplications (generic, squares, by small constants).

Selecting custom operators for software applications. Although the selection of FMA and DP2 can be quite trivial, taking full advantage of custom operators in final applications still requires a specific effort to ensure the best possible code generation during compilation. Furthermore, creating paired and specialized operators exploits the knowledge of the relationships and the ranges of the arguments, for which various compilation optimizations and heuristics on different levels of intermediate representations must be applied.

Particularly, for some specialized operators (like FSA, for example), an important condition is the positivity of floating-point operands for which the detection involves static range analysis techniques. Static integer range analysis has already been studied for a diverse range of compiler optimizations [Pat95a, BGS00, Sim08, SGPB11] and even for hardware customization [MRS+01]. However, it is a new try to explore floating-point domain by diverting integer analysis techniques.

This thesis addresses the issue of accelerating embedded applications via the algorithmic design and implementation of efficient custom floating-point operators as well as the development of compiler optimizations for their selection in application codes.

1.2 Contributions

1.2.1 Fused, specialized, and paired operators

We have designed and implemented in standard ANSI C11 a set of custom floating-point operators consisting of 4 fused operators, 5 specializations, and 2 paired operators. These operators, which are reviewed below, have been optimized for IEEE 754 single precision (called 'binary32 format' in [IEE08]) and fully support subnormal numbers, signed zeros, signed infinities, and NaNs (Not-a-Numbers). Correct rounding is provided for all of them except for trigonometric functions, which for cost reasons are implemented with a guaranteed accuracy of only one ulp (unit in the last place). Furthermore, correctly-rounded results can be obtained for any of the four rounding modes required by the standard for binary floating-point arithmetic: to nearest even (RN), up (RU), down (RD), and to zero (RZ).

Fused operators. We have implemented the following four fused operators. Each of them commits just one rounding error.

- **FMA** (fused multiply-add): $xy + z$.
- **FSA** (fused square-add): $x^2 + z$ with $z \geqslant 0$.
- **DP2** (two-dimensional dot product): $xy + zt$.
- **SOS** (sum of two squares): $x^2 + z^2$.

The FMA belongs to the IEEE 754 standard since its 2008 revision [IEE08, §5.4.1] and is a key operation for linear algebra and schemes like Horner's rule for evaluating polynomials and Newton's method for performing divisions. However, due to its lack of symmetry, this operator is well-known to introduce subtle programming issues when evaluating expressions like DP2; see [Kah96] and [Hig02, §2.6]. This is why we provide a correctly-rounded DP2 operator as well. We also provide FSA and SOS, which appear in n-dimensional and 2-dimensional Euclidean norm calculations; in those cases the algorithms for FMA and DP2 can be simplified significantly, thus providing opportunities for acceleration.

Specialized operators. We have also implemented the following five special cases of multiplication and addition:

- **mul2** (multiplication by two): $2x$.
- **div2** (multiplication by one half): $\frac{1}{2}x$.
- **scaleB** (multiplication by an integer power of two): $x \cdot 2^n$ with n a 32-bit signed integer.
- **square** (squaring): x^2.
- **addnn** (addition of nonnegative terms): $x + y$ with $x \geqslant 0$ and $y \geqslant 0$.

All these patterns appear in application codes and can be implemented much faster than generic multiplication or generic addition by means of specific algorithms. They are also fully specified by the IEEE 754 standard in the sense that mul2, div2, square, and addnn inherit the specification of multiplication and addition, and also since scaleB is itself specified in [IEE08, §5.3.3]. Furthermore, since we assume radix 2 floating-point arithmetic, no rounding occurs for mul2 and scaleB with $n \geqslant 0$, assuming no overflow occurs. Finally, FSA and SOS can of course be also considered as specialized versions of the fused operators FMA and DP2.

Paired operators. Two paired operators have been implemented so far, which typically occur in DSP kernels (FFT butterflies and rotations):

- **addsub** (simultaneous addition and subtraction): $(x + y, x - y)$.
- **sincos** (simultaneous sine and cosine over a reduced range): $(\sin x, \cos x)$ with $x \in [-\frac{\pi}{4}, \frac{\pi}{4}]$.

Such blocks give the opportunity to share some computations and also to expose more ILP in an easy way, thus allowing reduced latencies than by calling the two operations in sequence.

Design methodology. Our design aims at high instruction-level parallelism (ILP) as well as scalability to various floating-point formats.

To achieve high ILP, our first step is to classify the input into two categories, namely *generic* input and *special* input. Hence, the design of each operator reduces to three independent tasks: the evaluation of the condition which distinguishes between generic and special input (T_1), the handling of generic input (T_2) and that of special input (T_3). These three tasks can be processed simultaneously on a VLIW architecture, and the return value can be obtained by a selection between the results from T_2 and T_3 depending on the outcome of T_1. Our second step is to design high-ILP algorithms for each of these three tasks, assuming unbounded parallelism. However, T_1, T_2, T_3 are not considered separately and, following [BJJL+10], we optimize the *a priori* most expensive task first, and then try to reuse as much as possible its intermediate results for the other two tasks.

Although we have optimized our implementations for the binary32 format, all our algorithm descriptions (except for sincos) are *parametrized* by a more generic format, binary$k$, and accompanied with rigorous mathematical proofs. A first advantage of this approach is that it allows us to support other formats, like binary64 (double precision) or binary128 (quad precision), as soon as we have software support for 64-bit or 128-bit integer arithmetic. In fact, to facilitate engineering, we have described all these parametrized designs in an XML-based scheme, that is used to generate the C codes of all variants for different formats and rounding modes. Another advantage is to provide increased confidence into the produced C codes. Indeed, although we have performed exhaustive tests for univariate operators in binary32, for multivariate operators or for binary$k$ with $k \geqslant 64$, we had to resort to random tests and tests on well-chosen special cases. Parametrized algorithms, whose correctness proofs are done once for all, can therefore be seen as complementary of this classical validation scenario.

Operator performances for the binary32 format on the ST231. Table 1.1 gives for all rounding modes and the first 10 custom operators the latencies (in numbers of cycles) and code sizes (in numbers of integer instructions, and displayed within square brackets) obtained for one call on the ST231.

	RN		RU		RD		RZ	
mul2	5	[11]	7	[13]	6	[14]	6	[15]
div2	6	[16]	8	[17]	7	[16]	6	[13]
scaleB	15	[51]	16	[55]	16	[54]	14	[45]
square	12	[42]	11	[37]	9	[31]	9	[31]
addnn	15	[47]	15	[43]	14	[35]	14	[35]
FSA	22	[73]	22	[70]	19	[54]	19	[54]
FMA	46	[170]	46	[167]	45	[166]	42	[158]
SOS	26	[81]	25	[77]	22	[62]	22	[62]
DP2	55	[207]	54	[206]	53	[205]	51	[194]
addsub	28	[96]	30	[106]	30	[106]	26	[86]

Table 1.1: Performances for the binary32 format on the ST231 in # cycles [# instructions].

For sine and cosine over the reduced range $[-\pi/4, \pi/4]$ the following latencies and code sizes have been achieved:

sine	cosine	sincos
19 [31]	18 [28]	19 [49]

First, a good way to measure the efficiency of the custom operators compared to their naive implementation[1] is by evaluating speedups and code reduction ratios (CRR):

$$\text{speedup} = \frac{\text{latency of the original code}}{\text{latency of the improved code}}$$

and

$$\text{CRR} = \frac{\text{size of the improved code}}{\text{size of the original code}}.$$

In this way, speedups > 1 denote an acceleration, while CRRs < 1 indicate a code size reduction. Table 1.2 displays these ratios for rounding to nearest (except for sincos, which has 1-ulp accuracy), but the results are essentially the same for the three other rounding modes. From this table we see that speedups can be as high as 4.2, while CRRs can be as low as 0.15. Furthermore, for fused operators, since the speedups here do not account for the two function calls penalty, the exact performance gain obtained at run time can even be larger. Finally, it is worth noting that the FMA's adverse CRR is due to bigger alignment logic in the addition stage, which is necessary for correct rounding.

	Speedup	CRR
mul2	**4.2**	**0.15**
div2	3.5	0.22
scaleB	1.4	0.70
square	1.75	0.49
addnn	1.73	0.54
FSA	2.14	0.46
FMA	1.02	1.02
SOS	2.62	0.35
DP2	1.24	0.90
addsub	1.86	0.56
sincos	1.95	0.82

Table 1.2: Speedups and CRRs.

Second, a good measure of the exploitation of ILP is the Instruction Per Cycle ratio (IPC), which ideally should approach 4 for the 4-way VLIW ST231 processor. In practice, for sincos this ratio is $49/19 \approx 2.6$; for the other operators we see in Table 1.3 that their IPC values range from 1.9 (for the mul2 operator in RU) to 3.9 (for the DP2 operator in RD). On the other hand, the average IPC of these 11 operators is of 3.1, thus demonstrating high ILP exposure.

Third, to conclude this set of experiments Table 1.4 illustrates with the example of multiplication that naive specialization is not enough to obtain our results, and that entirely different algorithms have been needed. Here naive specialization means that 2, $\frac{1}{2}$, or x has been substituted to y in the C code of generic multiplication xy

[1] For example, the naive implementation of $xy + z$ consists of one generic multiplication followed by one generic addition; a naive implementation of sincos consists in one call to sine followed by one call to cosine.

	RN	RU	RD	RZ
mul2	2.2	**1.9**	2.3	2.5
div2	2.7	2.1	2.3	2.2
scaleB	3.4	3.4	3.4	3.2
square	3.5	3.4	3.4	3.4
addnn	3.1	2.9	2.5	2.5
FSA	3.3	3.2	2.8	2.8
FMA	3.7	3.6	3.7	3.8
SOS	3.1	3.1	2.8	2.8
DP2	3.8	3.8	**3.9**	3.8
addsub	3.4	3.5	3.5	3.3

Table 1.3: IPCs for the binary32 format on the ST231.

with, say, RN. After compilation of these new codes we see a latency reduction of at best 3 cycles.

Operation	Generic	Naive specialization	Optimized specialization
$2x$	21	18	5
$\frac{1}{2}x$	21	19	7
x^2	21	19	12

Table 1.4: Naive vs optimized specialization (in # cycles and for RN).

1.2.2 A production compiler supporting custom operators

Selection of custom operators. A specific variant of the production compiler of the ST200 C/C++ compiler has been developed to support the selection of custom floating-point operators. The selection of the fused operators (FMA, DP2) as well as most of the specialized operators (mul2, div2, scaleB of constant scaling factor, square, and SOS) can be achieved at high level, on mostly syntactic criteria, but as there are many choices to pattern-matching expressions, some heuristics are involved. The selection of paired operators (addsub, sincos) is more elaborate since the expressions candidate for such an association may not belong to the same statements. Finally, the selection of FSA and addnn requires an elaborate static analysis phase to prove the positivity condition of some of their arguments. The next three paragraphs detail how we have addressed these issues.

Squaring and scaling operators (mul2, div2 and $x \cdot C$ when C is a constant such that $C = 2^n$ and $n \neq \pm 1$) are specialized operators of general multiplication. In the production compiler, a floating-point multiplication can be selected as a square, when its two operands are identical; or selected as a scaling operator, when one of its operands is a constant power of two. Also, three kinds of patterns can be selected as DP2. According to the type of expressions x, y, z, and t, the expression $xy + zt$ is selected as `__dps`(x, y, z, t), which means single precision for dot product, or respectively, `__dpd`(x, y, z, t) for double precision. Similarly, $xy - zt$ is selected as `__dpsubs`(x, y, z, t) or `__dpsubd`(x, y, z, t), and $-xy - zt$ is selected as `__ndps`(x, y, z, t)

or `__ndpd`(x, y, z, t). Note finally that DP2 is selected as SOS when $x = y$ and $z = t$.

To select addsub, the compiler replaces all additions and subtractions by a call to a specific function returning a pair of floats. As addition and subtraction are *pure* functions known by the compiler not to have any side-effect, when both results are required for the same input, the specific function will be called only once after redundancy elimination. We use the same method to select sincos.

To select FSA and addnn, we need a specific optimization to prove the positivity of some operands during compilation. Since an integer range analysis framework is already implemented in the compiler, we realize this optimization by diverting it for floating-point variables.

Thanks to our production compiler, applications can benefit from the custom operators proposed so far. For example, the listing below shows how a radix-2 butterfly, which is a basic block for performing FFT, can be processed in our context. Here we see that the computation of `t1` and `t2` amounts to two DP2 fused operators, and the computations processed on array elements `x[k]`, `x[k+n1]`, and `y[k]`, `y[k+n1]` can be selected as two addsub paired operators. Therefore, the complete butterfly can be computed by using only custom operators. The paragraph below shows that significant performance gains are obtained in such applications by using the production compiler together with the complete set of custom operators.

```
for(k = j; k <= n; k = k + n2){
  t1 = c * x[k + n1] - s * y[k + n1];   →   t1 = __dpsubs(c, x[k+n1], s, y[k+n1])
  t2 = s * x[k + n1] + c * y[k + n1];   →   t2 = __dps(s, x[k+n1], c, y[k+n1])
  x[k + n1] = x[k] - t1;
                                        →   (x[k+n1], x[k]) = __adspairs(x[k], t1)
  x[k] = x[k] + t1;
  y[k + n1] = y[k] - t2;
                                        →   (y[k+n1], y[k]) = __adspairs(y[k], t2)
  y[k] = y[k] + t2; }
```

Performances on the UTDSP benchmark. The UTDSP Benchmark Suite [Lee] was created to assess C compilers efficiency on typical DSP codes. It is divided into two classes: *kernels* (FFTs, filters,...) and *applications* (LPC coding,...). It is also highly representative of our application domain and has been found, notably on compiler optimization work, to be a good predictor of the kind of improvements that can be obtained at a larger scale, on actual multimedia applications, such as audio coders and decoders.

Table 1.5 summarizes the gains on various UTDSP kernels and applications, which are:

- **FFT-256** (resp. **FFT-1024**), a complex radix-2 decimation-in-time 256-point (resp. 1024-point) FFT;
- **Latnrm-8** (resp. **Latnrm-32**), an 8th order (resp. 32nd) normalized lattice filter processing 1 (resp. 64) point(s);
- **SPE**, a power spectral estimator using the periodogram averaging method;
- **ADPCM**, an Adaptive Differential PCM encoder;

- **LPC**, a Linear Predictive Coding encoder.

Table 1.5 has been built by using internal compiler options that enable the selection of FMA operators only, and options that enable the selection of the full set of custom operators. The measures of the application performance are done with a cycle-accurate simulator, configured not to account for the I- or D- cache cycles: we call these 'perfect cycles'. For the floating-point operators, the D-cache cycles are zero by design, and the I-cache cycles are rapidly amortized for these relatively small functions in floating-point intensive code. For these test suites, the rounding mode is not changed and defaults to RN.

	FLIP 1.0	FMA		custom operators	
FFT-256	324001	304545	[1.06]	204194	[1.59]
FFT-1024	1609927	1512647	[1.06]	1010887	[1.59]
Latnrm-8	1857	1644	[1.13]	1388	[1.34]
Latnrm-32	467365	411173	[1.14]	347685	[1.34]
SPE	1186501	1116659	[1.06]	959924	[1.24]
ADPCM	1733237	1612096	[1.08]	1559928	[1.11]
LPC	874210	757948	[1.15]	747758	[1.17]

Table 1.5: Custom operators vs FLIP 1.0 in # cycles and [speedups].

Although the selection of the FMA alone brings speedups from 1.06 to 1.14 only, enabling our full set of custom operators brings speedups that can be as high as 1.59. The usage of such operators, that replace FMA or complement it, is displayed in Table 1.6. In this table "$X(Y\%)$" indicates that the proportion, over all available floating-point operators, of the custom operator X is of $Y\%$. As a consequence, a sum equal to 100% means that only custom operators are selected; this is for example the case of the FFT and Latnrm-32 benchmarks. Such benchmarks thus provide quantitative evidence of the benefits of selecting custom operators.

	custom operators selected
FFT-256,1024	DP2 (50%), addsub (50%)
Latnrm-8	DP2 (67%), FMA (29%)
Latnrm-32	DP2 (66%), FMA (34%)
SPE	DP2 (20%), FMA (13%), addsub (9%), SOS (5%), square (1%)
ADPCM	FMA (25%), DP2 (8%), square (4%)
LPC	FMA (72%), DP2 (4%), square (2%), addsub (<1%)

Table 1.6: Custom operators selected.

1.3 Thesis outline

The rest of the manuscript is organized into eight chapters as follows. Chapters 2 and 3 are preliminary chapters introducing, respectively, IEEE binary floating-point arithmetic and our design methodology on the ST231. The next four chapters detail our algorithms and implementations for a representative subset of six custom operators: some *specialized* products (square, scaleB, mul2, div2), the *fused* operator DP2, and the *paired* operator sincos. Then, Chapter 8 covers the techniques we have developed in the compiler for selecting custom operators in application codes. We conclude in Chapter 9.

IEEE binary floating-point arithmetic (Chapter 2). This chapter provides the necessary background in binary floating-point arithmetic according to the IEEE 754-2008 standard. There we review binary floating-point data, the parameters on which they depend, and how they are encoded into unsigned integers. We also recall the notions of rounding, overflow, gradual underflow, as well as a formula for the encoding of the rounded value of a real number. Finally, we present the function ulp (unit in the last place) together with some of its properties that will be useful later in this thesis.

Design principles for software floating-point on the ST231 (Chapter 3). In this chapter, we introduce the key architectural features of the ST231 for the floating-point software support, such as predicated execution through select operations, count-leading-zero instructions and encoding of immediate operands up to 32 bits. For a VLIW processor, the goal is to expose high instruction-level parallelism (ILP), and then we explain how to classify the inputs and define different tasks during the floating-point emulation to achieve high ILP. This is illustrated on the example of multiplication by two (mul2 operator). Finally, we detail the 64-bit integer support on the ST231 and how to generate C code for binary$k$ floating-point format from our parameterized implementations as long as the k-bit integer arithmetic is supported.

Squaring (Chapter 4). This chapter focuses on the squaring function $x \mapsto x^2$. We show how the specific properties of squaring can be exploited in order to design and implement algorithms that have much lower latency than those for general multiplication, while still guaranteeing correct rounding. Our algorithm descriptions are parameterized by the floating-point format, aim at high instruction-level parallelism (ILP) exposure, support subnormal numbers, and cover all rounding modes. We show further that their C implementation for the binary32 format yields efficient codes for targets like the ST231, with a latency at least 1.75x smaller than that of general multiplication in the same context.

Some parts of the work in this chapter have been published in [JJLMR11].

Scaling by integer powers of two (Chapter 5). Here we consider floating-point scaling, with subnormal support and correct rounding for all standard specified modes. By scaling, we mean multiplication by an integer power of two: given a floating-point datum x and an integer n, we want $x \cdot 2^n$. Depending on the sign of n, either overflow or inexact result may happen, which leads to very different algorithms. Therefore, we first separate the discussion for nonnegative and negative

n, and then we propose a complete implementation by simply merging the two cases. On a VLIW processor like the ST231, this method results in a low latency and high ILP scaling operator. Moreover, we present very efficient implementations for some specific values of n, such as 1 and -1, which indeed compute $2x$ and $x/2$ and can be considered as special cases of multiplication. The numerical results show that it is worthwhile to have dedicated algorithms for different values of n and that this basic operator can even impact some high-level applications like matrix balancing.

Two-dimensional dot products (Chapter 6). Various real applications require the evaluation of floating-point two-dimensional dot products $xy + zt$. In this chapter, we study how to evaluate such expressions accurately and efficiently on VLIW integer processors. Accurately means that we provide correct rounding for the all the rounding modes as well as support for subnormal numbers; efficiently means that it shall be faster than evaluating the expressions by the naive approach consisting of two multiplications followed by one addition. For this, we propose an algorithm and its correctness analysis, which, like for the previous two chapters, is done in a parametrized way. We also detail the corresponding C implementation for the binary32 format. On the ST231, this code is from 1.15x to 1.3x faster than the naive approach. It also exposes a lot of ILP, with an IPC of at least 3.8. Furthermore, combining it with other custom operators leads to significant speedups: 1.59x when performing FFT and up to 1.45x for some 3D graphics applications.

Simultaneous sine and cosine (Chapter 7). Graphics and signal processing applications often require that sines and cosines be evaluated at a same floating-point argument, and in such cases a very fast computation of the pair of values is desirable. In this chapter, we study how to exploit the 32-bit VLIW integer architecture of the ST231 in order to perform this task accurately for IEEE single precision, including subnormals. We describe software implementations for sinf, cosf, and sincosf over $[-\pi/4, \pi/4]$ that have a proven 1-ulp accuracy and whose latency on the ST231 is 19, 18, and 19 cycles, respectively. Such performances are obtained by introducing a novel algorithm for simultaneous sine and cosine that combines univariate and bivariate polynomial evaluation schemes.

Most of the work in this chapter has been published in [JJL12].

Compiler optimizations (Chapter 8). In this chapter, we focus on compilation aspects for the support of high-performance floating-point arithmetic on integer processors. First, to allow applications to benefit from custom floating-point operators, we study the selection of specialized, fused, and paired operators, that can be done at target-independent intermediate representation (WHIRL). Then, as the compiler is used to generate code for its own operators residing in a library (libgcc), we dedicate a specific effort at code generator intermediate representation (CGIR) level to ensure the best possible code selection when compiling the implementations of each floating-point operator on the ST231. Finally, for operators requiring to prove the positivity of some of their operands to be selected, such as fused square-add, we show how to augment the integer range analysis framework available at the CGIR level to detect this condition for floating-point variables. Compiling the UTDSP benchmark by the production compiler, we observe high usage of custom floating-point operators with speedups up to 1.59x.

Conclusions, perspectives, and appendices. In Chapter 9 we present our conclusions for both the computer arithmetic aspect and the compiler optimizations aspect, as well as some possible directions for future research. There are also three appendices: the first one summarizes the notation used throughout the document; the second and third ones provide C codes for some basic integer functions and for some compiler optimizations, respectively.

Chapter 2

IEEE binary floating-point arithmetic

This short preliminary chapter provides the necessary background in binary floating-point arithmetic according to the IEEE 754-2008 standard [IEE08]. We review binary floating-point data, the parameters on which they depend, and how they are encoded into unsigned integers. We also recall the notions of rounding, overflow, gradual underflow, as well as a formula for the encoding of the rounded value of a real number. Finally, we present the function *ulp* (unit in the last place) together with some of its properties that will be useful later in this thesis.

2.1 Introduction

Radix-β, precision-p floating-point numbers are numbers of the form

$$m \cdot \beta^e,$$

where the integer β is the radix of the floating-point system, the rational m is the significand and satisfies $|m| < \beta$ and $m \cdot \beta^{p-1} \in \mathbb{Z}$, and the integer e is the exponent. Kahan's paper, *Why Do We Need a Floating-Point Standard?* [Kah81], depicts the messy situation of floating-point arithmetic before the 1980s. The IEEE 754 Standard, first published in 1985, was proposed in order to address this issue and especially to improve the robustness, efficiency, and portability of programs used for numerical computations. Its latest revision dates back to August 2008, and is referred to as the IEEE 754-2008 Standard for Floating-Point Arithmetic [IEE08]. In this document, it is simply called the *standard*. Today, and as a result of this standardization effort, IEEE floating-point arithmetic is the most common way of approximating arithmetic with real numbers on a computer.

The standard defines floating-point formats, rounding modes, and operators, which are used to represent a finite subset of real numbers. The standard defines in particular various floating-point formats and their encodings into unsigned integers, rounding modes, exceptions and how to handle them, and what the result of an operation must be (for some given input/output formats and a given rounding mode).

Specifically, the standard defines two families of formats: *binary* ($\beta = 2$) and *decimal* ($\beta = 10$) floating-point formats. In this document, we will consider only *binary* floating-point formats, that is, we assume

$$\beta = 2.$$

For comprehensive reviews of floating-point arithmetic, we refer to the books [Hig02, §2] and [MBdD$^+$10], as well as to Goldberg's survey [Gol91].

Outline. We start in §2.2 by reviewing binary floating-point data, the parameters on which they depend, and how they are encoded into unsigned integers. Then §2.3 recalls the notions of rounding, overflow, gradual underflow, and provides a formula for the encoding of the rounded value of a real number. Finally, §2.4 presents the function ulp (unit in the last place) together with some of its properties that will be useful later in this thesis.

2.2 Binary floating-point data

2.2.1 Data defined by the standard

As specified in the standard [IEE08], binary floating-point data consist of

- quiet and signaling not-a-numbers (qNaN, sNaN),
- signed zeros ($+0$, -0),
- signed infinities ($+\infty$, $-\infty$),
- and finite nonzero binary floating-point numbers.

Finite nonzero binary floating-point numbers have the following form: given a precision p and an exponent range $[e_{\min}, e_{\max}]$,

$$x = (-1)^{s_x} \cdot m_x \cdot 2^{e_x}, \tag{2.1a}$$

where s_x is either 0 or 1, and where, writing $(\ldots)_2$ for a binary expansion,

$$m_x = (m_{x,0}.m_{x,1} \ldots m_{x,p-1})_2 \quad \text{and} \quad e_{\min} \leqslant e \leqslant e_{\max}. \tag{2.1b}$$

Here, since the radix is 2, each m_i is a bit, so that another way of writing m_x is

$$m_x = \sum_{i=0}^{p-1} m_{x,i} \cdot 2^{-i}.$$

The expression in (2.1a) with m_x and e_x as in (2.1b) is also called the *normalized representation* of a floating-point number [MBdD$^+$10, p. 15]. Any such number must in fact be

- either *normal* ($m_0 = 1$),
- or *subnormal* ($m_0 = 0$ and $e = e_{\min}$).

Thus, writing α for the smallest positive subnormal number and Ω for the largest normal number, we have

$$\alpha = 2^{e_{\min}-p+1} \qquad \text{and} \qquad \Omega = (2 - 2^{1-p}) \cdot 2^{e_{\max}}.$$

Remark. In this thesis, we will sometimes use the expression in (2.1) to represent signed zeros. In this case, we shall set e_x to $e_{\min}$ and set all the bits of m_x to zero.

Unbounded normalized representation. For our designs, we will often use the so-called *unbounded normalized representation* for intermediate results. This alternative representation to (2.1) is defined as follows. Let λ_x be the number of leading zeros of the binary expansion of m_x:

$$m_x = (\underbrace{0.00\dots00}_{\lambda_x \text{ zeros}}1m_{x,\lambda_x+1}\dots m_{x,p-1})_2.$$

Then

$$x = (-1)^{s_x} \cdot m'_x \cdot 2^{e'_x}, \tag{2.2}$$

where

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

This representation is normalized in the sense that m'_x is always in the real interval $[1, 2)$, whereas m_x can be either in $(0, 1)$ or in $[1, 2)$. It is unbounded, since now e'_x can be less than $e_{\min}$ and, therefore, outside the bounded exponent range $[e_{\min}, e_{\max}]$ of e_x.

Remark. When x is a normal number, $\lambda_x = 0$, and when x is a subnormal number, $\lambda_x = e_{\min} - e'_x > 0$.

Binary$k$ format. We shall assume that $e_{\max}$, $e_{\min}$, and p in (2.1b) satisfy

$$e_{\max} = 2^{w-1} - 1 \qquad \text{for some positive integer } w, \tag{2.3a}$$

and

$$e_{\min} = 1 - e_{\max}, \qquad 2 \leqslant p < e_{\max}. \tag{2.3b}$$

The format used to represent such floating-point data is called the *binaryk* format, where

$$k = w + p.$$

In particular, the standard specifies three basic binary floating-point formats: binary32, binary64, and binary128. Their parameters w and p are shown below together with the corresponding values of $e_{\min}$ and $e_{\max}$.

	k	w	p	$e_{\min}$	$e_{\max}$
binary32	32	8	24	-126	127
binary64	64	11	53	-1022	1023
binary128	128	15	113	-16382	16383

Table 2.1: Basic standard binary floating-point formats.

Property 2.1. *All the binaryk formats of the standard satisfy* (2.3a) *and* (2.3b)*.*

Proof. See [IEE08, Table 3.5] for the first two identities. On the other hand, the fact that $2 \leqslant p < e_{\max}$ for all standard binary formats can be shown as follows.

The standard binary formats are defined in [IEE08, Table 3.5]. There we see that, on one hand, this is true when $k \leqslant 128$. On the other hand, when $k > 128$ we have

$$p = k - j_k + 13 \quad \text{and} \quad e_{\max} = 2^{j_k - 14} - 1,$$

with $j_k = \text{round}(4 \log_2 k)$, the integer closest to $4 \log_2 k$. (If k is an integer then $4 \log_2 k$ cannot be exactly halfway two consecutive integers, so that no tie can occur.) By definition of round, $4 \log_2 k - 1/2 < j_k < 4 \log_2 k + 1/2$. This implies first that, for $k > 2^7$, $2 \leqslant p \leqslant k$. Second, $2^{-14.5} k^4 - 1 < e_{\max}$. Third, $k^3 > 2^{15.5}$ and then $2^{-14.5} k^4 > 2k > k + 1$, so that $k < e_{\max}$. □

Notice that the binary$k$ format is naturally expressed in terms of the parameters w and p. In our designs, we will heavily use this property, by providing algorithm descriptions parametrized by the format, that is, by w and p. This approach has the advantage of making specialization to a given format easy. For example, most of the C codes shown in this thesis for the binary32 format have been produced by setting

$$w = 8 \qquad \text{and} \qquad p = 24$$

in some parametrized formulas.

Here and hereafter, the set of binary$k$ finite floating-point numbers will be denoted by $\mathbb{F}$. We call *normal range* of $\mathbb{F}$ the set of real numbers ρ such that $2^{e_{\min}} \leqslant |\rho| \leqslant \Omega$.

2.2.2 Standard encoding into integers

Encoding of finite nonzero floating-point numbers. For the binary$k$ format, the standard encoding of x in (2.1) is via a k-bit unsigned integer X whose bit string (which we write $[X_{k-1} X_{k-2} \cdots X_0]$ and which we identify to X) satisfies

$$X = [s_x | E_{x,w-1} \cdots E_{x,0} | m_{x,1} \cdots m_{x,p-1}], \qquad E_x = \sum_{i=0}^{w-1} E_{x,i} 2^i, \tag{2.4}$$

with biased exponent

$$E_x = e_x - e_{\min} + m_{x,0}.$$

From (2.4) we deduce the following alternative equation for X:

$$X = s_x \cdot 2^{k-1} + D_x \cdot 2^{p-1} + M_x, \tag{2.5}$$

where

- s_x is the sign bit of x,
- $M_x = m_x \cdot 2^{p-1}$ is its *integer significand*,
- and $D_x = e_x - e_{\min}$ is a biased exponent of e_x.

Note that the bias for E_x is $-e_{\min}+m_{x,0}$, while it is $-e_{\min}$ for D_x. In our designs, (2.4) will be mainly used to extract some information from the input, and (2.5) will be mainly used to reconstruct the output.

In this document, we shall indicate that a floating-point datum x is encoded by X using the following notation:

$$X = \iota(x).$$

For example, for the binary32 number $x = -5/2$, we have $s_x = 1$, $m_x = 1 + 1/4 = (1.01)_2$, $e_x = 1$, so that $X = \iota(x) = 2^{31} + 127 \cdot 2^{23} + (1.01)_2 \cdot 2^{23} = 3223322624$.

Examples of extracting information from these encodings.

$\boxed{\text{Encoding of } |x|.}$ Let x be a finite floating-point number encoded by integer X. Then the encoding of its absolute value $|x|$ is clearly

$$X \bmod 2^{k-1} = [0|E_{x,w-1} \cdots E_{x,0}|m_{x,1} \cdots m_{x,p-1}].$$

Therefore, in the rest of this document, we shall use the following notation:

$$|X| = X \bmod 2^{k-1}.$$

Remark. When x is a subnormal number, the bit string of $|X|$ satisfies

$$\begin{aligned} |X| &= [\underbrace{00\ldots00}_{w+1} m_{x,1} \ldots m_{x,p-1}] \\ &= [\underbrace{00\ldots00}_{w+1}\underbrace{00\ldots00}_{\lambda_x-1} 1 m_{x,\lambda_x+1} \ldots m_{x,p-1}]. \end{aligned} \tag{2.6}$$

$\boxed{\text{Computing } \lambda_x.}$ Let x be a finite floating-point number. Recall that the number λ_x of leading zeros of the binary expansion of m_x is zero if x is normal, and as in (2.6) if x is subnormal. Then we have

$$\lambda_x = \max(\operatorname{clz}|X|, w) - w,$$

where the function clz counts the leading zeros of a k-bit unsigned integer Y:

$$\operatorname{clz} Y = \begin{cases} k, & \text{if } Y = 0, \\ k - 1 - \lfloor \log_2 Y \rfloor, & \text{otherwise.} \end{cases}$$

Encoding of special data. Zeros, infinities, and signaling or quiet NaNs are encoded via special values of X given in the table below. The bit of X of weight 2^{p-2} is zero for signaling NaNs and one for quiet NaNs.

In fact, as we have already remarked: when x is zero, we can write it as

$$x = (-1)^{s_x} \cdot m_x \cdot 2^{e_x}, \qquad \text{with} \quad m_x = 0 \quad \text{and} \quad e_x = e_{\min}.$$

Consequently, (2.4) can also be used for the encoding $\iota(0)$ of zero, which in some implementations may simplify the way zero is handled.

floating-point datum x	Value or range of integer X
$+0$	0
-0	2^{k-1}
$+\infty$	$2^{k-1} - 2^{p-1}$
$-\infty$	$2^{k} - 2^{p-1}$
sNaN	$\|X\| \in [2^{k-1} - 2^{p-1} + 1, 2^{k-1} - 2^{p-2} - 1]$
qNaN	$\|X\| \in [2^{k-1} - 2^{p-2}, 2^{k-1} - 1]$

Table 2.2: Standard encoding of special data.

Another consequence of the encodings above is that for finite floating-point numbers and special values, the integer E_x in (2.4) satisfies

$$E_x = \begin{cases} e_x - e_{\min} + m_{x,0}, & \text{if } x \text{ is a finite floating-point number,} \\ 2^w - 1, & \text{if } x \text{ is } \pm\infty \text{ or NaN.} \end{cases} \tag{2.7}$$

Therefore, we see in particular that $E_x = 0$ if and only if x is zero or subnormal.

2.3 Rounding

In general, the result of an operation (or a function) on floating-point numbers is not exactly representable in the floating-point system being used, so it has to be rounded. The standard has brought out the notion of *rounding mode*: how a numerical value is rounded to a finite (or, possibly, infinite) floating-point number is specified by a rounding mode (also called rounding-direction attribute in the 2008 revision of the standard [IEE08, pp. 16]).

2.3.1 IEEE rounding modes

The four rounding modes that are specified in the standard are:

- Round to nearest: RN(x) is the floating-point number that is the closest to x. To ensure uniqueness when x falls exactly halfway between two consecutive floating-point numbers, the 2008 revision of the standard specifies two tie-breaking rules: round to nearest *even* and round to nearest *away*. Since this second rule is not required by binary implementations, the tie-breaking rule chosen in this document is to *even*: x is rounded to the only one of these two consecutive floating-point numbers whose integer significand is even. For example, consider the rational number $x_0 = 16777217/134217728$. Here x_0 is halfway between the two consecutive binary32 floating-point numbers $x_0' = 2^{-3}$ and $x_0'' = (1 + 2^{-23}) \cdot 2^{-3}$, whose integer significands are 2^{23} and $2^{23} + 1$. Therefore, RN(x_0) $= x_0'$.

- Round toward $+\infty$ (also known as 'round up'): RU(x) is the floating-point number (possibly $+\infty$) closest to and no less than x; for example, RU(x_0) $= x_0''$.

- Round toward $-\infty$ (also known as 'round down'): $\mathrm{RD}(x)$ is the floating-point number (possibly $-\infty$) closest to and no greater than x; for example, $\mathrm{RD}(x_0) = x_0^{'}$.

- Round toward zero: $\mathrm{RZ}(x)$ is the floating-point number closest to and no greater in magnitude than x; it is equal to $\mathrm{RD}(x)$ if $x \geqslant 0$, and to $\mathrm{RU}(x)$ if $x \leqslant 0$; for example, $\mathrm{RZ}(x_0) = x_0^{'}$.

Rounding modes have many properties: see for instance [MBdD+10, §2.2]. In particular, RN and RZ are odd functions, that is,

$$\mathrm{RN}(-x) = -\mathrm{RN}(x) \qquad \text{and} \qquad \mathrm{RZ}(-x) = -\mathrm{RZ}(x), \tag{2.8a}$$

while RU and RD are related to each other as follows:

$$\mathrm{RU}(-x) = -\mathrm{RD}(x) \qquad \text{and} \qquad \mathrm{RD}(-x) = -\mathrm{RU}(x). \tag{2.8b}$$

2.3.2 Overflow

Overflow occurs when the rounded result with an unbounded exponent range would have an exponent larger than $e_{\max}$. More precisely, given the unbounded normalized representation of a nonzero real number x,

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

where $m_x \in [1, 2)$ and e_x unbounded, we have

- overflow before rounding when $e_x > e_{\max}$;

- overflow after rounding when $e_x = e_{\max}$ and $m_x > 2 - 2^{1-p}$.

Whatever overflow before rounding or after rounding, the standard specifies [IEE08, §7.4] that

- round to nearest carries all overflows to ∞ with the sign of the intermediate result;

- round toward zero carries all overflows to Ω with the sign of the intermediate result;

- round toward $+\infty$ carries negative overflows to $-\Omega$, and carries positive overflows to $+\infty$;

- round toward $-\infty$ carries positive overflows to Ω, and carries negative overflows $-\infty$.

2.3.3 Gradual underflow

The availability of subnormal numbers allows for what Kahan calls *gradual underflow* (also called *graceful underflow*): the loss of precision is slow instead of being abrupt. Several interesting properties are valid only when subnormal numbers are available: for example, if $x \neq y$ then the computed value of $x - y$ is nonzero, or if x and y are finite floating-point numbers such that $\frac{y}{2} \leqslant x \leqslant 2y$ then $x - y$ is exactly representable [Kah96, Hau96, Ste74]. Moreover, the support of gradual underflow can significantly ease the writing of stable numerical software [Dem84].

Although it is considered the most difficult type of numbers to implement in floating-point units in hardware [SSDT05], software emulation is not necessarily costly. For example, on ST231, the native hardware instruction `clz` counts the leading zeros of a 32-bit integer in only 1 cycle, which leads to the efficient detection of subnormal inputs. Furthermore, we will see in the rest of the document (especially in Chapters 4 and 7) that for some particular operators, such as square, sine, and cosine, subnormal support can be obtained for free after careful analysis.

2.3.4 Encoding of the rounded value of a real number

Let $\circ$ denote one of the four rounding modes: $\circ \in \{\mathrm{RN}, \mathrm{RU}, \mathrm{RD}, \mathrm{RZ}\}$. The standard integer encoding of a rounded real number is specified by the fact below [Rev09, §2.3] on the condition that overflow before rounding does not occur.

Fact 2.1. *Let ρ be a real number such that*

$$\rho = (-1)^s \cdot \ell \cdot 2^d, \tag{2.9}$$

where $s \in \{0, 1\}$ and where $(\ell, d) \in \mathbb{R} \times \mathbb{Z}$ satisfies

- *either $\ell \in [0, 1)$ and $d = e_{min}$,*
- *or $\ell \in [1, 2)$ and $d \in [e_{min}, e_{max}]$.*

Then its standard encoding of the rounded value $\circ(\rho)$ is the integer R such that

$$R = s \cdot 2^{k-1} + D \cdot 2^{p-1} + L + b, \tag{2.10a}$$

where the three integers D, L and b satisfy

$$D = d - e_{min}, \tag{2.10b}$$

$$L = \lfloor \ell \cdot 2^{p-1} \rfloor, \tag{2.10c}$$

$$b = \begin{cases} g \wedge (\ell_{p-1} \vee t) & \text{if } \circ = RN, \\ (g \vee t) \wedge (\neg s) & \text{if } \circ = RU, \\ (g \vee t) \wedge s & \text{if } \circ = RD, \\ 0 & \text{if } \circ = RZ, \end{cases} \tag{2.10d}$$

and where

$$\ell_{p-1} = L \bmod 2, \qquad g = \lfloor \ell \cdot 2^p \rfloor \bmod 2, \qquad t = \big[\{\ell \cdot 2^p\} \neq 0\big].$$

Here and hereafter, for any two integers $i, j \in \{0,1\}$, we write $i \wedge j$, $i \vee j$, $\neg i$ for, respectively, the integers ij, $i+j-ij$, $1-i$. In addition, throughout this document, we shall call b the *round bit*, g the *guard bit*, and t the *sticky bit*. Writing $\ell = (\ell_0.\ell_1 \dots \ell_{p-1}\ell_p \dots)_2$, we have in particular

$$g = \ell_p \qquad \text{and} \qquad t = \ell_{p+1} \vee \ell_{p+2} \vee \cdots . \tag{2.11}$$

When ρ is known to be positive, for example $\rho = x^2$ with $x \neq 0$, the computation of the rounding bit b is somehow simpler. By applying $s = 0$, Equation (2.10d) is simplified to

$$b = \begin{cases} g \wedge (\ell_{p-1} \vee t) & \text{if } \circ = \mathrm{RN}, \\ g \vee t & \text{if } \circ = \mathrm{RU}, \\ 0 & \text{if } \circ = \{\mathrm{RD}, \mathrm{RZ}\}. \end{cases} \tag{2.12}$$

For example, consider $\rho = 16777217/134217728$. For the binary32 format, we have $s = 0$, $d = -3$,

$$\ell = (1.\underbrace{00\dots00}_{23}1)_2$$

and thus $D = 123$ and $L = 2^{23}$. It follows that $\ell_{p-1} = 0$, $g = 1$, $t = 0$, and then $b = 0$ if $\circ \in \{\mathrm{RN}, \mathrm{RD}, \mathrm{RZ}\}$ and $b = 1$ if $\circ = \mathrm{RU}$.

The expressions in (2.12) are in fact classical (see for example [EL04, §8.4.3]), and to prove Fact 2.1, it suffices to combine them with the properties of the rounding modes in (2.8) and with Equation (2.5).

Note that for overflow after rounding, Fact 2.1 ensures the right encoding of the result.

2.4 The ulp function

Units in the last place (ulps) are often used to indicate the accuracy of floating-point results. Given a floating-point system of precision p and minimal exponent $e_{\min}$, the ulp of any *real* number x can be defined as follows [Mul05, MBdD+10]:

$$\mathrm{ulp}(x) = \begin{cases} 0 & \text{if } x = 0, \\ 2^{\max\{e_{\min}, e\} - p + 1} & \text{otherwise,} \end{cases}$$

with e the integer power of two such that $2^e \leqslant |x| < 2^{e+1}$. For example, for $x = 16777217/134217728 = 0.1250000075\dots$, we have $2^{-3} \leqslant x < 2^{-2}$, so that $e = -3$; for the binary32 format, this gives $\mathrm{ulp}(x) = 2^{\max\{-126,-3\}-24+1} = 2^{-26}$.

When $\circ = \mathrm{RN}$, we have

$$|\circ(x) - x| \leqslant \frac{1}{2}\mathrm{ulp}(x). \tag{2.13}$$

For other rounding modes, $\circ \in \{\mathrm{RU}, \mathrm{RD}, \mathrm{RZ}\}$, we have

$$|\circ(x) - x| \leqslant \mathrm{ulp}(x). \tag{2.14}$$

In (2.13) we say that an accuracy of “half an ulp” is achieved, while in (2.14), we talk about 1-ulp accuracy. Similarly to (2.14), in Chapter 7 we will ensure 1-ulp accuracy for our sine and cosine operators: this means that the result r returned by any of these two operators satisfies

$$|r - \rho| \leqslant \mathrm{ulp}(\rho),$$

where ρ is the exact result.

Relationship with unit roundoff u. Another quantity used very frequently in numerical error analysis is the so-called *unit roundoff* u defined in terms of the precision p as

$$u = 2^{-p}.$$

When the real number x is such that $|x| \in [2^{e_{\min}}, \Omega]$, we have $2^e \leqslant |x|$ and, on the other hand, $\mathrm{ulp}(x) = 2^{e-p+1}$, so that

$$\mathrm{ulp}(x) \leqslant 2u|x|.$$

Hence, in particular,

$$|x| \in [2^{e_{\min}}, \Omega] \qquad \Rightarrow \qquad \frac{|\mathrm{RN}(x) - x|}{|x|} \leqslant u.$$

Some basic properties. We conclude this chapter with some basic properties of the ulp function, that are easy to check and will be useful in the sequel:

- For any real x,
$$\mathrm{ulp}(x) = \mathrm{ulp}(|x|). \tag{2.15a}$$

- For $x, y \in \mathbb{R}$,
$$|x| \leqslant |y| \quad \Rightarrow \quad \mathrm{ulp}(x) \leqslant \mathrm{ulp}(y). \tag{2.15b}$$

- If $j \in \mathbb{Z}$ and $x \in \mathbb{R}$ are such that both x and $2^j x$ are in the normal range of $\mathbb{F}$ then
$$\mathrm{ulp}(2^j x) = 2^j \mathrm{ulp}(x). \tag{2.15c}$$

- For any real x in the normal range of $\mathbb{F}$, we have
$$2^{1-p} \leqslant |x|\mathrm{ulp}(x) < 2^{2-p}. \tag{2.15d}$$

- For $x \in \mathbb{R}$,
$$0 < |x| < 2^{e_{\min}+1} \quad \Rightarrow \quad \mathrm{ulp}(x) = 2^{e_{\min}-p+1} =: \alpha. \tag{2.15e}$$

Chapter 3

Design principles for software floating-point support on the ST231

The ST231 is a 4-way 32-bit very long instruction word (VLIW) integer processor from STMicroelectronics widely used in embedded media processing systems. In this chapter, we start in §3.1 by introducing the key architectural features of the ST231 for the floating-point software support, such as predicated execution through select operations, count-leading-zero instructions and encoding of immediate operands up to 32 bits. For a VLIW processor, the goal is to expose high instruction-level parallelism (ILP), and we explain in §3.2 how to classify the inputs and define different tasks during the floating-point emulation to achieve high ILP. This is illustrated on the example of multiplication by two (mul2 operator) in §3.3. Finally, we detail in §3.4 the 64-bit integer support on the ST231 and how to generate C code for binary$k$ floating-point format from our parameterized implementations as long as the k-bit integer arithmetic is supported.

3.1 Introduction to the ST231 and its C/C++ compiler

3.1.1 Architecture overview

The ST231 is a 4-way 32-bit VLIW integer core from STMicroelectronics' ST200 family derived from the Lx technology platform [FBF+00], highly used in audio and video domains, such as HD-IPTV, set-top-boxes, printers, wireless terminals, etc.

The architecture of the ST231 (depicted in Figure 3.1) includes the following features:

- parallel execution units, including multiple integer ALUs and two 32×32-bit multipliers;

- a large register file of 64 32-bit general purpose registers (`$r0`[2] to `$r63`[3]) and 8 1-bit condition registers (`$b0` to `$b7`);

- predicated execution through 'select' operations;

- efficient branch architecture with multiple condition registers;

[2] Reading `$r0` always returns the value zero. Writing to `$r0` has no effect on the processor state.

[3] `$r63`, the architectural link register, is used by the call and return mechanism.

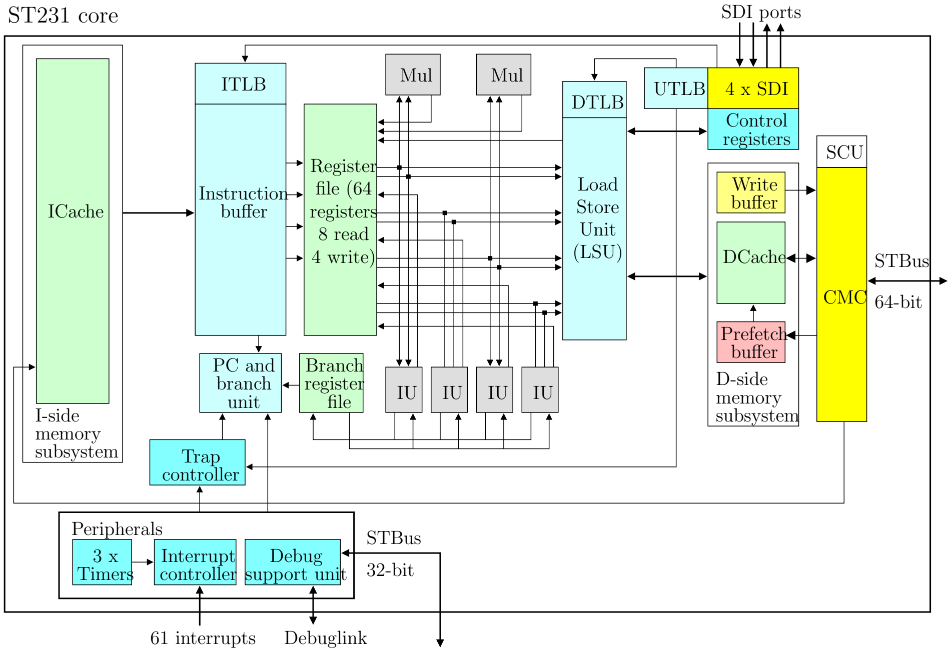


Figure 3.1: Architecture of the ST231 processor core.

- encoding of immediate operands up to 32 bits.

The two 32×32-bit multipliers can be used simultaneously and the execution is fully pipelined, that is, we can start two multiplications every cycle. Together with the feature that immediate values up to 32 bits can be encoded in one instruction, fast polynomial evaluation for elementary functions such as sine and cosine can be implemented efficiently on the ST231, as detailed in Chapter 7.

Except for multiplication whose latency is 3 cycles, the latency of all the other integer instructions, such as addition and subtraction, is only 1 cycle.

In addition to the usual operations, the ALUs also implement several specific 1-cycle instructions dedicated to code optimization: *leading zero count* (which is key to subnormal numbers support and mantissa alignment), computation of *minimum* and *maximum*, arbitrary left and right *shifts*, and combined *shift and add*.

Table 3.1 summarizes the ST231 instructions which are frequently used in our floating-point emulation and how to use them in C.

Signed and unsigned maximum and minimum instructions can be selected by the compiler from the C expressions in Table 3.1. For better readability of our implementations, we wrap such ternary operators into an inline function. For example, we define the `minu` function as follows, which will be used in Listing 3.5 in § 3.3. Other functions are detailed in Appendix A.

```
inline uint32_t minu(uint32_t x, uint32_t y){
 return (x<y)?x:y;}
```

Description	ST231 instruction	C expression (X, Y: int32_t or uint32_t)
addition, subtraction	add, sub	X+Y, X−Y
add with carry and generate carry	addcg	__st200addcg(X,Y,C), see §3.1.5
32 × 32 unsigned product	mul32, mul64hu	(uint64_t)X *(uint64_t)Y
bitwise AND	and	X & Y
logical AND	andl	X && Y
count leading zeros	clz	__lzcntw(X), see § 3.1.5
bitwise OR	or	X | Y
logical OR	orl	X || Y
(un)signed maximum	max(u)	(X > Y) ? X : Y
(un)signed minimum	min(u)	(X < Y) ? X : Y
shift N bits and add, N ∈ {1,2,3,4}	shNadd	(X ≪ N) + Y
shift left, shift right	shl, shr(u)	X ≪ N, X ≫ N
bitwise XOR	xor	X ^ Y
select instructions	slct	if (cond) {...} else {...}

Table 3.1: Elements of the ST231 instruction architecture set.

3.1.2 Instruction bundling and extended immediates

As a 4-way VLIW processor, the ST231 issues simultaneously up to 4 instructions, which are encoded into a wide word, called a bundle. 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.

Except for `clz`, all the other instructions in Table 3.1 can operate on registers as well as on immediates. For example, for unsigned minimum, `minu`, we have the two following forms:

- Register form.

  ```
  minu $dest = $src1, $src2
  ```

  Instruction `minu` applies to registers `$src1` and `$src2`.

- Immediate form.

  ```
  minu $dest = $src1, ISRC2
  ```

  Instruction `minu` applies to registers `$src1` and immediate value `ISRC2`.

In general, only small (9-bit) immediates can be directly encoded in a single word syllable. In the event that larger immediates are required, an "immediate extension" is used. This extension is encoded in an adjacent word in the bundle, making the operation effectively a two-word operation. These immediate extensions associate either with the operation to their left or their right in the bundle.

Though this reduces available instruction parallelism, it is an effective mechanism to build large constants, avoiding any access through the data memory, as we will see in the next subsection.

3.1.3 Data access in memory

The core accesses the memory system through two separated L1 caches: a 32-Kbyte 4-way associative data cache (D-cache), and a 32-Kbyte direct-mapped instruction cache (I-cache).

The direct mapped organization of the I-cache creates a specific difficulty to obtain good and reproducible performance, since its caching performance is very sensitive to the code layout, that can create conflict misses. This has been addressed by post link time optimizations [GRBB05] that are either driven by heuristics or by actual code profile.

If the datum is in the D-cache, it is directly accessed by the *Load Store Unit* (LSU), and the load-use latency is 3 cycles. If the datum is not cached, the latency is dependent on the actual implementation of the memory sub-system: controller and memory type. In this case, loading a cache line into the D-cache may cost up to 100 cycles. If the datum is present in the prefetch buffer, it is transferred first to the D-cache, which may take around 10 cycles, and then loaded by the LSU. In the code designed for this thesis, the choice was made not to use any in-memory table, to avoid any access through the D-cache side of the system, because a cold miss entails a significant performance hit and hoisting the prefetch early enough to be efficient is impossible due to various barriers, mostly function calls.

3.1.4 ST200 VLIW compiler

The ST200 compiler, st200cc, is based on the Open64 technology,[4] open-sourced by SGI in 2001, further developed by STMicroelectronics and more generally by the Open64 community.

The Open64 compiler has been re-targeted to support different variants of the ST200 family of cores by a dedicated tool, called the Machine Description System (MDS), providing an automatic flow from the architectural description to the compiler and other binary utilities.

Though the compiler has been developed in the beginning to achieve high performance on embedded media C code, it has been further developed and is able to compile a fully functional Linux kernel and distribution, including, for instance, C++ graphics applications based on Web Kit.

It is organized as follows: the gcc-4.2.0 based front-end translates C/C++ source code into a first high level target independent representation called WHIRL, that is further lowered and optimized by the middle-end, including WHIRL global optimizer (WOPT), based on static single assignment form (SSA) representations, and optionally loop nest optimizer (LNO). It is then translated in a low-level target dependent representation, code generator intermediate representation (CGIR) for code generation, including code selection, low level loop transformations, if-conversion, scheduling, and register allocation.

In addition the compiler is able to work in a specific interprocedural analysis (IPA) and interprocedural optimization (IPO) mode where the compiler builds a representation for a whole program, and optimizes it globally by constant propagation, inlining, code cloning, and other optimizations.

[4] http://www.open64.net

Several additions have been done by STMicroelectronics to achieve high performance goals for the ST200 target:

- A dedicated linear assembly optimizer (LAO) is in charge at the CGIR level of software pipelining, pre-pass and post-pass scheduling. It embeds a nearly optimal scheduler based on an integer linear programming (ILP) formulation of the pipelining problem [ADN07]. As the problem instances are very large, a large neighborhood search heuristic is applied as described in [dD07] and the ILP problem is further solved by an embedded GLPK (GNU Linear Programming Kit) solver.
- A specific if-conversion phase, designed to transform control flow into 'select' operations [Bru09].
- Some additions to the CGIR 'extended block optimizer' (EBO), including a dedicated 'range analysis' and 'range propagation' phase.
- A proved and efficient out-of-SSA translation phase, including coalescing improvements [BDR+09].

Besides efficient code selection, register allocation, and instruction scheduling, the key optimizations contributing to the generation of the low-latency floating-point software are mostly the if-conversion optimization, and to a lesser extent the range analysis framework.

3.1.5 ST231 intrinsics

The st200cc compiler recognizes a number of intrinsic operators (also called builtin functions) which can be used to produce assembly language statements that otherwise can not be expressed through standard ANSI C/C++.

Table 3.2 gives the ones frequently used for floating-point emulation:

Intrinsic prototype	ST231 operation
`uint32_t __lzcntw(uint32_t)`	`clz`
`uint32_t __lzcntl(uint64_t)`	`clz`
`uint64_t __st200addcg(uint32_t, uint32_t, uint32_t)`	`addcg`

Table 3.2: ST231 intrinsic functions frequently used for floating-point emulation.

Count leading zeros. Intrinsic operator `__lzcntw` simply uses the hardware instruction `clz` to count the leading zeros of a 32-bit integer. However, we implement a C function `uint32_t clz(uint32_t)` in order to make our implementations portable to other architectures.

```
inline uint32_t clz(uint32_t x){
  #ifdef __ST200__
    return __lzcntw(x);
  #else
```

```
    uint32_t n;
    if ( x == 0 ) return 32;
    n = 0;
    if ( x <= 0x0000FFFF ) { n = n + 16; x = x << 16; }
    if ( x <= 0x00FFFFFF ) { n = n +  8; x = x <<  8; }
    if ( x <= 0x0FFFFFFF ) { n = n +  4; x = x <<  4; }
    if ( x <= 0x3FFFFFFF ) { n = n +  2; x = x <<  2; }
    if ( x <= 0x7FFFFFFF ) { n = n +  1; }
    return n;
  #endif
}
```

We are aware that GNU C[5] also provides the built-in functions to count the leading zeros of an integer, such as `__builtin_clz`, `__builtin_clzl`, and `__builtin_clzll`. As the result of zero input is not defined for such functions, the length of the integer format should be returned explicitly for our floating-point emulation. For example, to implement `clz` by using `__builtin_clz`, we must write the following code.

Listing 3.1: `clz` by using GNU C built-in.

```
inline uint32_t clz(uint32_t x){
  if ( x == 0 ) return 32;
  else return __builtin_clz (x);
}
```

On the ST231, whether by `__lzcntw` or by Listing 3.1, this function is implemented by the hardware instruction listed below. Register `$r16` holds the input `x` and meanwhile the result is also written to the same register.

```
  clz   $r16=$r16                          ## (cycle 0)
```

The implementation for `__lzcntl`, which counts the leading zeros of a 64-bit integer, is detailed in § 3.4.1.

Remark. In floating-point arithmetic, we often need to compute λ_x, which is defined as the number of leading zeros of the significand m_x of a finite floating-point number x. For the binary32 format, λ_x can be implemented efficiently on the ST231 by using `clz` as follows.

```
lambda = maxu (clz (X & 0x7FFFFFFF), 8) - 8;
```

Addition with carry and generate carry. `__st200addcg` adds two unsigned 32-bit integers together with a 1-bit input carry, whose implementation is listed below.

```
        convib  $b0=$r18                   ## (cycle 0)
         ;; ## (bundle 0)
```

[5] http://gcc.gnu.org/

```
        addcg   $r16, $b0=$r16, $r17, $b0        ## (cycle 1)
         ;; ## (bundle 1)
        convbi  $r17=$b0                 ## (cycle 2)
        return  $r63                     ## (cycle 2)
         ;; ## (bundle 2)
```

Registers `$r16` and `$r17` hold the two input integers, `$r18` holds the input carry and the 64-bit result is written back to `$r16` (lower 32 bits) and `$r17` (upper 32 bits).

According to the ST231 run-time architecture manual [ST209], argument values up to 32 bytes are passed in register `$r16` to `$r23` and arguments beyond these registers appear in memory. Return values up to 32 bytes also appear in these registers.

3.2 “Multi-tasks” for high instruction-level parallelism

3.2.1 Predicated execution for conditional branches reduction

The architecture supports predicated execution through *select* operations to improve performance by removing conditional branches. The *select* semantic is

```
 $dest = slct $cond, $src1, $src2
```

Here `$cond` is a condition register and it usually holds the result of a comparison instruction. When `$cond` is true, the `$dest` register takes the value in `$src1`; otherwise, it takes the value in `$src2`.

In st200cc, the if-conversion optimization is implemented to eliminate conditional branches by transforming a control flow region into an equivalent set of conditional instructions [Bru09].

Example of if-conversion. Listing 3.2 gives a piece of C code that returns -1 if $x < 0$, and 1 otherwise.

Listing 3.2: Example of if-conversion.

```
int32_t exp_ifconversion(int32_t x){
  if( x<0 ) return -1;
  else      return  1;}
```

By using st200cc, Listing 3.3 gives the assembly code without if-conversion, and 3.4 with if-conversion. Listing 3.3 shows that the critical path of this function takes 5 cycles when the branch is generated. The reason is that there is a 3-cycle latency after an operation writing the condition register (`$b0` at line 0) to issue the operation branch (`brf` at line 3).

Listing 3.3: The ST231 assembly code with branches.

```
0       cmpge   $b0=$r16, $r0                    ## (cycle 0)
1       mov     $r16=1                           ## (cycle 0)
2        ;; ## (bundle 0)
3       brf     $b0, .L_BB2_exp_ifconversion     ## (cycle 3)
```

```
          ;; ## (bundle 1)
        return  $r63                              ## (cycle 4)
          ;; ## (bundle 4)
.L_BB2_exp_ifconversion:
        mov     $r16=-1                           ## (cycle 0)
        return  $r63                              ## (cycle 0)
          ;; ## (bundle 0)
```

Listing 3.4 shows that the critical path takes 2 cycles by using *slct*. At line 0, condition register `$b0` obtains the result whether variable `x` stored in register `$r16` is negative or not. At line 4, the `slct` instruction writes the final result to register `$r16` depending on the value of `$b0`. There is no latency restriction between `cmpge` and `slct` operation.

Listing 3.4: The ST231 assembly code with if-convention.

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

3.2.2 If-conversion in emulating floating-point arithmetic

Let us now present the methology on how to expose high ILP for floating-point arithmetic by using if-conversion.

For each operator, we classify inputs as being either **special** or **generic** by introducing the following definitions:

- **special**: the inputs are such that the output is a value known in advance, like a constant ($+0$, $-\infty$, $\ldots$), or the input itself.
- **generic**: every input that is not special.

Using Knuth's bracket notation

$$[\mathcal{S}] = \begin{cases} 1, & \text{if } \mathcal{S} \text{ is true,} \\ 0, & \text{if } \mathcal{S} \text{ is false,} \end{cases}$$

we define the condition

$$C_{\text{spec}} = [x \text{ is special}]. \tag{3.1}$$

This condition allows us to reach the following high-level algorithmic description, which shows that implementation of each operator essentially reduces to three independent tasks T_1, T_2, and T_3:

```
evaluate the condition $C_{\text{spec}}$          [$T_1$]
if ($C_{\text{spec}} = 1$) then
    handle special inputs                 [$T_2$]
else
    handle generic inputs                 [$T_3$]
```

Thanks to if-conversion, the generated assembly code for the above algorithm will consist of a straight-line piece of code computing the result R_i of each task T_i and ending with a 'slct' instruction that selects R_2 if R_1 is true, R_3 otherwise.

For the design and implementation of each task we shall proceed in two steps as in [BJJL+10]: assuming unbounded parallelism we optimize the *a priori* most expensive task first, namely task T_3 and then only T_1 and T_2, by trying to reuse as much as possible what was computed for T_3. The latency of 'slct' being 1 cycle, the lowest latency we can expect for any operator is thus 1 cycle more than that of T_3.

3.3 An example of binary32 implementation exposing high ILP

In this section, we show how the method introduced in the previous section (§3.2.1) can be applied in a real design in order to expose high ILP. The example we have chosen here is the implementation for the binary32 format of **mul2**, which multiplies a floating-point datum x by two. The detailed algorithm will be discussed in Chapter 5, and the goal here is simply to give a general view of how to use the architectural support of the ST231 (integer arithmetic, if-conversion, etc.) in order to realize floating-point emulation.

Defining generic and special input of mul2. The **mul2** operator is a specialization of general multiplication. When $\circ = \mathrm{RN}$, it is specified by the standard as

$$r = \begin{cases} +\infty & \text{if } x \geqslant 2^{e_{\max}}, \\ \circ(2x) & \text{if } |x| < 2^{e_{\max}}, \\ -\infty & \text{if } x \leqslant -2^{e_{\max}}, \\ \mathrm{qNaN} & \text{if } x \text{ is NaN}. \end{cases} \tag{3.2}$$

Then, we say the floating-point number x is **generic** if $|x| \in [0, 2^{e_{\max}})$, and **special** otherwise. Therefore, C_{spec}, defined by Equation (3.1), satisfies

$$\begin{aligned} C_{\text{spec}} = \quad & \big[x \text{ is } \pm\infty \text{ or NaN}\big] \\ & \vee \big[x \text{ is finite number and } |x| \geqslant 2^{e_{\max}}\big]. \end{aligned} \tag{3.3}$$

Computing correctly-rounded multiplication by 2. Given x generic, we will see in Chapter 5 that the binary32 format encodings of x and r satisfy

$$R = X + \min(|X|, 0x00800000), \quad |X| = X \bmod 0x80000000.$$

The corresponding C implementation is shown at line 10 of Listing 3.5.

Detecting and handling special input for $\circ = $ RN. Let us now consider the implementation of the condition C_{spec}, which satisfies

$$C_{\text{spec}} = C_{\text{large}} \vee C_{\text{nan}}, \tag{3.4}$$

where $C_{\text{large}} = [|x| \geqslant 2^{e_{\max}}]$ and $C_{\text{nan}} = [x \text{ is NaN}]$. Therefore, the C implementation for the binary32 format is

$$C_{\text{large}} \vee C_{\text{nan}} = \big[|X| \geqslant 0x7F000000\big].$$

According to the specification shown by Equation (3.2), the C implementation below gives the handling of special input for the binary32 format. Here, `absX` at line 0, which computes the encoding of $|x|$, is used in both T_1 (evaluation of C_{spec}) and T_3 (generic input handling).

Listing 3.5: Implementation of **Mul2** for *binary32*, $\circ = $ RN.

```
 0 uint32_t mul2(uint32_t X){
 1   uint32_t absX,sigX,Cspec,Cnan;
 2   absX  = X & 0x7FFFFFFF;       //shared in T1 and T3
 3   Cspec =  absX >= 0x7F000000; //evaluate the condition Cspec, T1
 4   if(Cspec){                   //handling special input, T2
 5     sigX = X & 0x80000000;
 6     Cnan  = absX > 0x7F800000;
 7     if(Cnan)   return  X | 0x00400000;
 8     else       return  sigX | 0x7F800000;
 9   }else{
10     return     X + minu(absX, 0x00800000);   //computing the generic T3
11   }
12 }
```

ILP exposed on the ST231. In order to illustrate the ILP exposed by the C code in Listing 3.5 on the ST231, we present it in Listing 3.6 with a layout that gives an idea of the scheduling and latency of computing each variable.

The statements that can appear in the same line present the instructions able to be executed in one bundle; therefore, up to 4 instructions may appear in one line. The constants longer than 9 bits are considered as an instruction as well, since they occupy an additional syllable. Meanwhile, a variable appearing on the left side of the assignment symbol (=) at line i implies that this variable is evaluated at cycle i and will be available for use (appearing on the right side of the assignment symbol) at cycle $i+1$. The instructions that dominate the latency of the operator are usually the first instruction of each line. This in general gives a good idea of the length of the critical path.

We can see from Listing 3.6 that, unlike other operators in the next chapters, the handling of special inputs of **mul2** is on the critical path for the binary32 format and $\circ = $ RN.

Listing 3.6: Implementation of **mul2** for the binary32 format and $\circ = \mathrm{RN}$.

```
absX  = X & 0x7FFFFFFF;           sigX  = X & 0x80000000;
Rspec = sigX | 0x7F800000;        qnan  = X | 0x00400000;
Cnan  = absX > 0x7F800000;        delta = minu(absX, 0x00800000)
if(Cnan) Rspec = qnan;    Cspec = absX >= 0x7F000000;   Rgen = X + delta;
if(Cspec)R = Rspec;       else R = Rgen;         return R;
```

Listing 3.7 gives the assembly codes generated by the compiler. As we have explained in §3.1.2, the constants from line 0 to line 11 are 32-bit immediates and therefore each of them occupies an extra syllable. Thus, from cycle 0 to cycle 3, the 4 syllables of each bundle are fully used, which indicates the highest achievable use of ILP.

Listing 3.7: Implementation of **mul2** for the binary32 format and $\circ = \mathrm{RN}$.

```
 0        and     $r20=$r16, -2147483648           ## (cycle 0)
 1        and     $r18=$r16, 2147483647            ## (cycle 0)
 2         ;; ## (bundle 0)
 3        or      $r20=$r20, 2139095040            ## (cycle 1)
 4        or      $r19=$r16, 4194304               ## (cycle 1)
 5         ;; ## (bundle 1)
 6        cmpgtu  $b1=$r18, 2139095040     ## (cycle 2)
 7        minu    $r17=$r18, 8388608       ## (cycle 2)
 8         ;; ## (bundle 2)
 9        add     $r17=$r16, $r17                  ## (cycle 3)
10        slct    $r16=$b1, $r19, $r20     ## (cycle 3)
11        cmpgeu  $b0=$r18, 2130706432     ## (cycle 3)
12         ;; ## (bundle 3)
13        slct    $r16=$b0, $r16, $r17     ## (cycle 4)
14        return  $r63                     ## (cycle 4)
15         ;; ## (bundle 4)
```

3.4 Parameterized implementation for a specific binary$k$ format

We have optimized our algorithms for the binary32 format. However, as we have seen on the previous example of the **mul2** operator, all the analysis is carried out in a parameterized fashion, that is, expressed in terms of the IEEE 754 binary$k$ format. In fact, except for sine and cosine, all the algorithms presented in the rest of this document are parameterized by the format factors, such as p, w and k. Then, we describe our algorithms by Extensible Markup Language (XML) in a parameterized way and use Python scripts to decode them into C code for a specific format. The XML description is close to the mathematical expressions, and easy to check and to maintain. Therefore, the implementation for a specific format is direct as soon as the corresponding k-bit fixed-point arithmetic support is available. On the ST231, 32-bit integer support is native, while 64-bit and 128-bit integer supports are emulated.

For the implementations of sine and cosine, we have currently focused on the binary32 format, but some code for other formats could in principle be obtained too, by recomputing the polynomial approximations and their error analyzes.

In the following, we will first introduce the 64-bit and 128-bit integer supports on ST231. Then, we will present our XML-Python based mechanism to generate C code implementation for any specific binary*k* format.

3.4.1 64-bit and 128-bit integer support on the ST231

64-bit integer support. The ST231 compiler fully supports the C11 standard 'long long' type and its unsigned variant as a 64-bit integral type, emulated on the 32-bit architecture.

For every expression that contains an operation on such a 64-bit type (arithmetic, comparison, logical, conversions), the compiler has to generate emulation code, either in the form of a function call, when the operation is long, or in the form of direct assembly emission, that we call 'open code'. Obviously, logical operations and compares are emitted as 'open code' since the length of the sequence emitted is not significantly bigger that a function call. The conversion sequences are either pure register operations for integral type (for instance copies, or sign or zero extensions), but rely on calls to the compiler run-time support for conversions with floating-point types.

The choice was also done to emit 'open code' for most arithmetic operations (with the exception of division and modulus operators, implemented as function calls), even though the multiplication can incur a high code size.

One key benefit of emitting 'open code' is that the emission is done at a reasonably high level in the compiler code generator, and thus benefits from all further optimizations. This enables for instance to simplify 64-bit operations that can be statically detected as being in reality 32-bit only (whether for the high or low 32-bits), a frequent case in practice. Some further optimizations refined from our practice have been proposed in Chapter 8.

Here are some implementations of 64-bit integer support on the ST231:

- Unsigned minimum.

```
uint64_t minu64(uint64_t x, uint64_t y){
  return (x<y)?x:y;
}
```

```
        cmpltu  $b1=$r17, $r19           ## (cycle 0)
        cmpeq   $b0=$r17, $r19           ## (cycle 0)
        minu    $r8=$r16, $r18           ## (cycle 0)
        minu    $r17=$r17, $r19          ## (cycle 0)
         ;; ## (bundle 0)
        slct    $r16=$b1, $r16, $r18     ## (cycle 1)
         ;; ## (bundle 1)
        slct    $r16=$b0, $r8, $r16      ## (cycle 2)
        return  $r63                     ## (cycle 2)
```

```
         ;; ## (bundle 2)
```

- Unsigned left shift.

```
uint64_t shiftL(uint64_t x, uint32_t n){
  return x << n;
}
```

```
        sub     $r19=32, $r18                    ## (cycle 0)
        add     $r20=$r18, -32                   ## (cycle 0)
        shl     $r17=$r17, $r18                  ## (cycle 0)
         ;; ## (bundle 0)
        shru    $r19=$r16, $r19                  ## (cycle 1)
        shl     $r20=$r16, $r20                  ## (cycle 1)
        shl     $r16=$r16, $r18                  ## (cycle 1)
         ;; ## (bundle 1)
        or      $r18=$r19, $r20                  ## (cycle 2)
         ;; ## (bundle 2)
        or      $r17=$r17, $r18                  ## (cycle 3)
        return  $r63                     ## (cycle 3)
         ;; ## (bundle 3)
```

- Implementation of intrinsic call `__lzcntl`, which counts the leading zeros of a 64-bit integer.

```
        clz     $r16=$r16                        ## (cycle 0)
        clz     $r18=$r17                        ## (cycle 0)
        convib  $b0=$r17                         ## (cycle 0)
         ;; ## (bundle 0)
        add     $r16=$r16, 32                    ## (cycle 1)
         ;; ## (bundle 1)
        slctf   $r16=$b0, $r16, $r18             ## (cycle 2)
        return  $r63                             ## (cycle 2)
         ;; ## (bundle 2)
```

128-bit integer support. The 128-bit support exists only as a prototype library with no native implementation in the compiler and we implement 128-bit integer operations by function calls. Now, suffering from the memory access generated by the compiler due to the representation of 128-bit types by structures, it is not as efficient as it could be. Ideally, implementing the native support for the __uint128_t and __int128_t gcc extensions (at the moment limited to 64-bit architectures) would alleviate these issues.

Here is the implementation of counting leading zeros for a 128-bit integer:

```
typedef struct {
  uint64_t l;
  uint64_t h;
}uint128_t;
uint32_t clz128(uint128_t x){
   uint32_t n1 = __lzcntl(x.h);
   uint32_t n2 = __lzcntl(x.l);
   if(n1 == 64) return 64 + n2 ; else return n1 ;
}
```

```
        clz     $r20=$r18                        ## (cycle 0)
        clz     $r22=$r16                        ## (cycle 0)
        clz     $r21=$r19                        ## (cycle 0)
        convib  $b2=$r19                 ## (cycle 0)
         ;; ## (bundle 0)
        stw     0[$r12]=$r16                     ## (cycle 1)  x
        add     $r20=$r20, 32                    ## (cycle 1)
        add     $r16=$r22, 32                    ## (cycle 1)
        convib  $b1=$r17                 ## (cycle 1)
         ;; ## (bundle 1)
        stw     4[$r12]=$r17                     ## (cycle 2)  x+4
        slctf   $r20=$b2, $r20, $r21     ## (cycle 2)
        clz     $r21=$r17                        ## (cycle 2)
         ;; ## (bundle 2)
        stw     8[$r12]=$r18                     ## (cycle 3)  x+8
        slctf   $r16=$b1, $r16, $r21     ## (cycle 3)
        cmpeq   $b0=$r20, 64             ## (cycle 3)
         ;; ## (bundle 3)
        stw     12[$r12]=$r19                    ## (cycle 4)  x+12
        add     $r16=$r16, 64                    ## (cycle 4)
         ;; ## (bundle 4)
        slct    $r16=$b0, $r16, $r20     ## (cycle 5)
        return  $r63                     ## (cycle 5)
         ;; ## (bundle 5)
```

Table 3.3 gives the latencies of some integer operators for the 32-, 64-, and 128-bit formats. As the compiler does not support 128-bit integers, `mulh` for 64-bit integer, which returns the upper half of a 64×64 multiplication product, is also supported by function call. The detailed implementations for unsigned integer multiplications are in Appendix A.

	32-bit	64-bit	128-bit
unsigned add	1	3	9
unsigned sub	1	3	16
unsigned min,max	1	3	28
unsigned shifts, $\gg$,$\ll$	1	4	25
clz	1	3	6
mulh	3	8	29
mull	3	5	14
(un)signed >,>=,<,<=	1	2	10
(un)signed ==	1	2	9

Table 3.3: Performances of the integer arithmetic supports on the ST231 in # cycles.

3.4.2 XML-based implementation for various formats

XML basics. Extensible Markup Language (XML) is a markup language that defines a set of rules to encode documents in a flexible and readable way. XML 1.0 Specification is produced by the W3C [W3C]. The key constructs are:

- *Tags*, which begin with $<$ and end with $>$, are used either in pairs with start-tags and end-tags, or as empty-element tags (for example, $<$ empty-tag/ $>$).

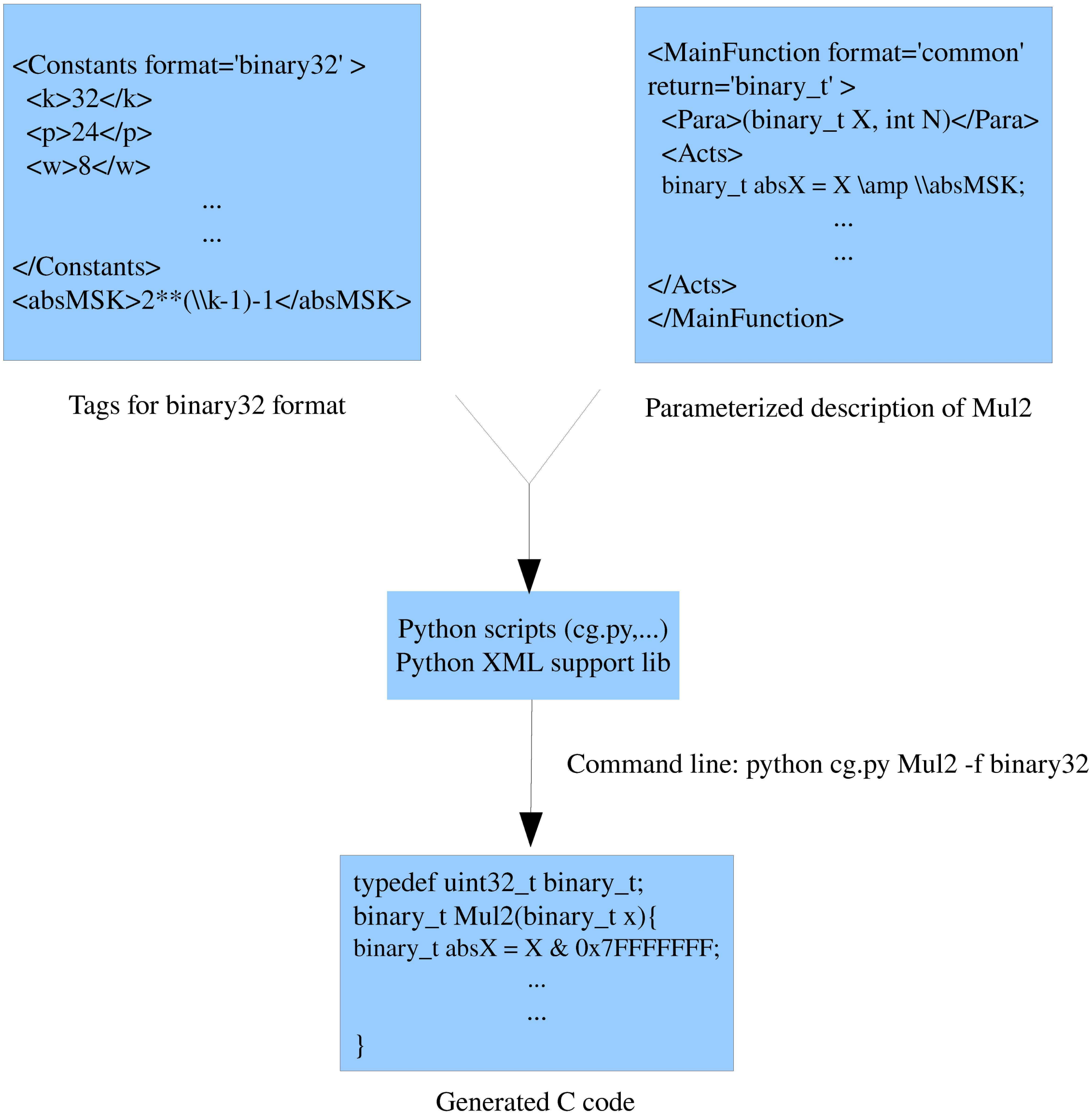
- *Attributes*, which are quoted either by single or double quotes, consist of name/value pairs that exist within a start-tag or empty-element tag.

An XML document consists of XML elements, such that an XML element is either everything from its start-tag to its end-tag or consists only of an empty-element tag. The listing below gives the example of an XML element. Here $<$ `INF format='binary32'` $>$ is a start-tag with name `INF` and with attribute `format`, whose value is `binary32`; $<$ `/INF` $>$ is an end-tag.

```
<INF format='binary32'> 0x7F800000 </INF>;
```

Overview of the code generation scheme. As our algorithms are fully parameterized, we separate our implementation into two parts: algorithm description on one hand, and integer arithmetic support on the other hand. In our design, the algorithms are described in a parameterized way in XML tags and meanwhile the format-specified constants such as k, w and p, and k-bit integer operators are also implemented in XML tags with an attribute dedicated to `binary32`, `binary64` and `binary128`.

Figure 3.2 gives an overview on how this works. We have developed some Python scripts to decode such XML files and to generate corresponding C codes by using some Python standard library tools such as an XML parser [Pyt].



```
<Constants format='binary32' >
  <k>32</k>
  <p>24</p>
  <w>8</w>
                  ...
                  ...
</Constants>
<absMSK>2**(\\k-1)-1</absMSK>
```

```
<MainFunction format='common'
return='binary_t' >
  <Para>(binary_t X, int N)</Para>
  <Acts>
  binary_t absX = X \amp \\absMSK;
                  ...
                  ...
</Acts>
</MainFunction>
```

Command line: python cg.py Mul2 -f binary32

```
typedef uint32_t binary_t;
binary_t Mul2(binary_t x){
binary_t absX = X & 0x7FFFFFFF;
                  ...
                  ...
}
```

Figure 3.2: XML-Python for C code generation.

This approach is essentially similar to the typical C conditional compilation directives, shown below.

```
#if defined (binary32)
  #define k 32
  #define p 24
...
#elif defined (binary64)
  #define k 64
  #define p 53
...
#else
...
#endif
```

However, as we have seen that the various k-bit integer supports can be different on the same processor, their C implementations will be slightly different.

For example, on the ST231, as there is no native support for 128-bit integer types in the compiler, we must use structures to represent 128-bit integer types and explicitly call functions for their integer operations, which is not the case for the support of 32- or 64-bit integer operators. Although C++ operator overloading would be a solution to write generic code, the final performance of the floating-point emulation may suffer from the object abstraction penalty.

In summary, our framework enables us to abstract the floating-point arithmetic description away from the implementation details such as the binary$k$ support. Except for sine and cosine, all the other implementations introduced in this thesis work are described by XML in our framework. However, since the support for the 128-bit integer operators is not complete yet, this should still be considered as work in progress.

Chapter 4

Squaring

This chapter focuses on the squaring function $x \mapsto x^2$. We show how the specific properties of squaring can be exploited in order to design and implement algorithms that have much lower latency than those for general multiplication, while still guaranteeing correct rounding. Our algorithm descriptions are parameterized by the floating-point format, aim at high instruction-level parallelism (ILP) exposure, support subnormal numbers, and cover all rounding modes. We show further that their C implementation for the binary32 format yields efficient codes for targets like the ST231, with a latency at least 1.75x smaller than that of general multiplication in the same context.

Some parts of the work in this chapter have been published in [JJLMR11].

4.1 Introduction

In this chapter we focus on the specialization of the multiplication operator $f : (x, y) \mapsto x \times y$ into a square operator $x \mapsto x^2$. Squares of floating-point values are ubiquitous in scientific computing and signal processing, since they are intensively used in any algorithm requiring the computation of Euclidean norms, powers, sample variances, etc. [Hig02, PTVF07].

We give a thorough study of how to design efficient software for IEEE floating-point squaring on targets like the ST231, that is, by means of integer arithmetic and logic only, and with high ILP exposure.

Our first contribution is to show how the specific properties of squaring can be exploited in order to refine the IEEE specification of multiplication, to deduce definitions of special input and generic input that are suitable for implementation, and to optimize the generic path and the special path for latency. This analysis is done for all rounding modes and presented in a parameterized fashion, in terms of the precision and the exponent range of the input/output floating-point format.

Second, this analysis allows us to produce a complete portable C code for the binary32 format and each rounding mode. On the ST231 processor, the result is a latency of 12 cycles for rounding 'to nearest even,' which is 1.75x faster than the 21 cycle latency of the multiply operator of FLIP 1.0; for the other rounding modes, the speedups are even higher and range from 1.9 to 2.3. Also, the average number of instructions issued every cycle lies between 3.4 and 3.5, thus indicating heavy use of the 4 issues available.

Third, we report on some experiments involving non-IEEE variants and square-intensive applications. We show that relaxing the IEEE requirements (finite math only, no support of subnormals) saves at most 1 cycle in the context of the ST231. We also show that for applications like the Euclidean norm, the sample variance, and binary powering the practical impact of our fast squarer reflects well the best that can theoretically be achieved.

For squaring in integer / fixed-point arithmetic several optimized hardware designs have been proposed; see for example [EL04, §4.9] as well as [WSS01, Gök08] and the references therein. However, for squaring in IEEE floating-point arithmetic much less seems to be available and to the best of our knowledge, no optimized design has been presented and analyzed in the details as we do in this chapter, be it in hardware or in software. Furthermore, the implementation of squaring for the binary32 format and the ST231 processor outlined in [Rai06] does not support subnormal numbers, is available only for rounding 'to nearest even,' and has a latency of 27 cycles, which is 2.25x more than our 12 cycle latency.

Outline. First, we show in §4.2 how to specialize to squaring the IEEE specification of multiplication, deduce suitable definitions of generic and special input, and give a high-level algorithmic view of the squaring operator. Then, §§4.3 and 4.4 detail our algorithms for handling, respectively, generic and special input by means of integer arithmetic, and give the corresponding C implementation for the binary32 format. The performances of this implementation and the improvements for real applications on the ST231 processor are reported in §4.5.

4.2 Specification

Squaring being a special case of general multiplication $x \times y$, it is fully specified by the IEEE 754-2008 standard: given a rounding mode $\circ$ and assuming $x = y$, the result r prescribed by [IEE08] for $x \times y$ is as follows:

$$r = \begin{cases} |x| & \text{if } x \in \{\pm 0, \pm\infty\}, \\ \text{qNaN} & \text{if } x \text{ is NaN}, \\ \circ(x^2) & \text{otherwise}. \end{cases} \tag{4.1}$$

This specification shows that r is essentially known in advance when x is zero, infinity, or NaN. However, in the particular case of squaring, if $|x|$ is nonzero but "small enough" then the rounded value $\circ(x^2)$ will always be equal to a tiny constant (0 or α). Similarly, if $|x|$ is finite but "large enough" then $\circ(x^2)$ will always be equal to a huge constant (Ω or $+\infty$). The rest of this section studies such properties of $\circ(x^2)$ in order to refine the specification (4.1) and deduce a high-level algorithm.

Let $\min_\circ$ and $\max_\circ$ denote, respectively, the minimum value and maximum value of $\circ(x^2)$ for $|x|$ in $[\alpha, \Omega]$. Using the monotonicity, for $x \geqslant 0$, of the map $x \mapsto x^2$ together with the definitions of rounding and overflow recalled in Chapter 2, one may check that these values are as follows:

$\circ$	RN	RD	RU
$\min_\circ$	$+0$	$+0$	α
$\max_\circ$	$+\infty$	Ω	$+\infty$

(4.2)

For which values of x are such extremal values attained? To answer this, let us define the following two quantities

$$\alpha' = 2^{\lfloor (e_{\min} - p)/2 \rfloor} \quad \text{and} \quad \Omega' = 2^{(e_{\max}+1)/2}. \tag{4.3}$$

We give below three properties regarding these quantities.

Property 4.1. *The values α' and Ω' defined in (4.3) are normal floating-point numbers such that $\alpha' < \Omega'$.*

Proof. Since α' and Ω' in (4.3) are integer powers of two it suffices to verify that

$$e_{\min} \leqslant \lfloor (e_{\min} - p)/2 \rfloor < (e_{\max} + 1)/2 \leqslant e_{\max}.$$

The leftmost inequality is equivalent to $e_{\min} \leqslant (e_{\min} - p)/2$ because $e_{\min}$ is an integer; since $e_{\min} = 1 - e_{\max}$, the latter inequality is itself equivalent to $p < e_{\max}$, which is true by (2.3b). From (2.3b) it also follows in particular that $e_{\max} \geqslant 1$, which implies the rightmost inequality. The remaining inequality follows from the fact that (2.3b) implies that $e_{\min}$ is negative while p and $e_{\max}$ are positive. □

Property 4.2. *For $\circ \in \{RN, RD, RU\}$ and x a finite nonzero floating-point number, one has $\circ(x^2) = \max_\circ$ iff $|x| \geqslant \Omega'$.*

Proof. If $|x| \geqslant \Omega'$ then $x^2 \geqslant 2^{e_{\max}+1}$, so that $\circ(x^2) = \max_\circ$ for each $\circ$. Conversely, assume that $|x| < \Omega'$. Since x is a finite floating-point number in precision p, we deduce that $|x| \leqslant (2 - 2^{1-p}) \cdot 2^{(e_{\max}-1)/2}$ and then $x^2 \leqslant C \cdot 2^{e_{\max}}$, with $C = 2(1 - 2^{-p})^2$. Since $C < 2 - 2^{1-p}$ one has further $x^2 < \Omega$. This implies $\circ(x^2) < \max_\circ$ for each $\circ$ and the conclusion follows. □

The interval $[\Omega', \Omega]$ thus defines the widest input range on which $\max_\circ$ is achieved. Interestingly, this range is the same for all our rounding modes, which makes things simpler from the implementer point of view. Note also that Ω' is an integer power of two because of (2.3a). For $\min_\circ$ the situation is slightly more complex, as this minimum value is achieved on an input range $[\alpha, \alpha'_\circ]$ whose upper bound now depends on $\circ$. However, the property below shows that one can suppress this dependency by restricting to input ranges whose upper bound has the form 2^i for some integer i.

Property 4.3. *The value α' in (4.3) is the largest integer power of two such that, for every finite nonzero floating-point number x in $[\alpha, \alpha')$, $\circ(x^2) = \min_\circ$ for $\circ \in \{RN, RD, RU\}$.*

Proof. If $\alpha \leqslant x < \alpha'$ then $0 < x^2 < \alpha/2$, so that $\mathrm{RN}(x^2) = \mathrm{RD}(x^2) = +0$ and $\mathrm{RU}(x^2) = \alpha$. This shows that $x \in [\alpha, \alpha')$ implies $\circ(x^2) = \min_\circ$ for all $\circ$. To prove the maximality of α' it suffices to check that $\mathrm{RN}(y^2) \neq \min_{\mathrm{RN}}$ for some floating-point number y in $[\alpha', 2\alpha')$, say $y = \frac{3}{2}\alpha'$. We have $y^2 = \frac{9}{4} 2^{2\lfloor (e_{\min}-p)/2 \rfloor}$ and, using the fact that $\lfloor i/2 \rfloor \geqslant (i-1)/2$ when $i \in \mathbb{Z}$, we deduce that $y^2 \geqslant \frac{9}{8}\alpha/2 > \alpha/2$. It follows that $\mathrm{RN}(y^2) = \alpha$, which differs from $\min_{\mathrm{RN}} = +0$. □

The main outcome of Properties 4.2 and 4.3 is the following specification of floating-point squaring, which refines (4.1):

$$r = \begin{cases} +0 & \text{if } x = \pm 0, \\ \min_\circ & \text{if } \alpha \leqslant |x| < \alpha', \\ \circ(x^2) & \text{if } \alpha' \leqslant |x| < \Omega', \\ \max_\circ & \text{if } \Omega' \leqslant |x| \leqslant \Omega, \\ +\infty & \text{if } x = \pm\infty, \\ \mathrm{qNaN} & \text{if } x \text{ is NaN.} \end{cases} \tag{4.4}$$

This brings a natural distinction between two kinds of input:

Definition 4.1. *Input x is called* **generic** *if $\alpha' \leqslant |x| < \Omega'$, and* **special** *otherwise.*

By Property 4.1 every subnormal input is special, so that generic input consist of normal numbers only. The corresponding output $\circ(x^2)$ must be finite because of the "only if" part in Property 4.2, but it can be (sub)normal or zero.

4.3 Computing correctly-rounded squares for generic input

In this section we consider the computation of $\circ(x^2)$ for x generic, that is, x as in (2.1) and such that

$$\alpha' \leq |x| < \Omega'. \tag{4.5}$$

By Property 4.1 such an x is normal, and thus $1 \leqslant m < 2$.

4.3.1 A normalized formula for x^2

Normalized representation of the exact square. A first step towards the computation of $\circ(x^2)$ consists in normalizing the representation $m^2 \cdot 2^{2e}$ of x^2 implied by (2.1a). Let

$$c = \left[m \geqslant \sqrt{2}\right]. \tag{4.6}$$

Defining $m' = m^2 \cdot 2^{-c}$ and $e' = c + 2e$ then yields the unique pair $(m', e') \in \mathbb{R} \times \mathbb{Z}$ such that $1 \leqslant m' < 2$ and

$$x^2 = m' \cdot 2^{e'}. \tag{4.7}$$

This is the so-called *unbounded normalized representation* of the exact square. Tight bounds for e' are given by the next result.

Property 4.4. *The normalized exponent e' of x^2 satisfies*

$$2\lfloor(e_{min} - p)/2\rfloor \leqslant e' \leqslant e_{max}.$$

Proof. Recalling (4.3) and taking squares in (4.5), we obtain $2^{2\lfloor(e_{\min}-p)/2\rfloor} \leqslant m' \cdot 2^{e'} < 2^{e_{\max}+1}$. On the one hand, $m' < 2$ implies $2\lfloor(e_{\min} - p)/2\rfloor < e' + 1$ and, since both sides are integers, the announced lower bound follows. On the other hand, $1 \leqslant m'$ implies $e' < e_{\max} + 1$ and, similarly, we deduce the announced upper bound. □

These bounds for e' are the best possible ones:

- The lower bound is attained when $|x| = \alpha'$. It is equal to $e_{\min} - p - \epsilon$ with $\epsilon = \left[p \text{ is odd}\right]$. One has $\epsilon = 0$ for the standard binary32 format, and $\epsilon = 1$ for the standard binary16, binary64, and binary128 formats [IEE08, §3.6].
- The upper bound $e_{\max}$ is attained for example when $x = (1.1)_2 \cdot 2^{(e_{\max}-1)/2}$, which satisfies (4.5) and whose square is $(1.001)_2 \cdot 2^{e_{\max}}$.

Also, the tight lower bound on e' is less than $e_{\min}$ for $p \geqslant 2$, so that both situations $e' \geqslant e_{\min}$ and $e' < e_{\min}$ do occur.

Correctly-rounded value of the exact square. When $e' \geqslant e_{\min}$ the normalized representation (4.7) already allows to express the correctly-rounded value $\circ(x^2)$. In this case x^2 lies in the range $[2^{e_{\min}}, 2^{e_{\max}+1})$ and, since $m' \in [1,2)$,

$$\begin{aligned} \circ(x^2) &= \circ(m') \cdot 2^{e'} \\ &= \circ(m^2 \cdot 2^{-c}) \cdot 2^{c+2e}. \end{aligned} \tag{4.8a}$$

When $e' < e_{\min}$ the exact square ranges in $(0, 2^{e_{\min}})$. In this case, we first set the exponent to $e_{\min}$ and only then round the resulting scaled significand in fixed point:

$$\begin{aligned} \circ(x^2) &= \tilde{\circ}(m' \cdot 2^{-(e_{\min}-e')}) \cdot 2^{e_{\min}} \\ &= \tilde{\circ}(m^2 \cdot 2^{-(e_{\min}-2e)}) \cdot 2^{e_{\min}}, \end{aligned} \tag{4.8b}$$

where $\tilde{\circ}$ denotes the function that rounds the reals from $[0,2)$ in the same direction as $\circ$ but on the regular grid $\{i \cdot 2^{1-p} : i = 0, 1, \ldots, 2^p\}$.

In order to handle the cases (4.8a) and (4.8b) simultaneously, let us define

$$\mu = \max(c, e_{\min} - 2e). \tag{4.9}$$

Then, the exact result of square is given in both cases by

$$x^2 = \ell \cdot 2^d, \tag{4.10a}$$

where

$$\ell = m^2 \cdot 2^{-\mu} \qquad \text{and} \qquad d = \mu + 2e. \tag{4.10b}$$

By construction we have

- either $\ell \in (0,1)$ and $d = e_{\min}$,
- or $\ell \in [1,2)$ and $d \in [e_{\min}, e_{\max}]$.

The property below further gives bounds for the range of μ.

Property 4.5. *$c \leqslant \mu \leqslant p + \epsilon$ with $\epsilon = \big[p$ is odd$\big]$.*

Proof. The lower bound is an immediate consequence of the definition of μ in (4.9). To establish the claimed upper bound we consider two cases:

- If $\mu = c$ then μ is at most 1 and cannot be larger than $p + \epsilon$, since $p \geqslant 2$ and $\epsilon \geqslant 0$.
- If $\mu = e_{\min} - 2e$ then, recalling that $e' = c + 2e$ and noticing that the lower bound in Property 4.4 equals $e_{\min} - p + \epsilon$ (because $e_{\min}$ is even), we obtain $\mu \leqslant p + \epsilon + c$. Since both μ and $p + \epsilon$ are even integers, and since c is either 0 or 1, it follows that $\mu \leqslant p + \epsilon$.

Hence $\mu \leqslant p + \epsilon$ in both cases, and the proof follows. □

Again, these bounds are tight: $x = 1$ gives $\mu = 0$, while $x = \pm\alpha'$ gives $c = 0$, $e' = e_{\min} - p - \epsilon$, and then $\mu = p + \epsilon$.

4.3.2 Implementation of $\circ(x^2)$ for the binary$k$ format

We detail here how to implement, for x generic and the binary$k$ floating-point format, the computation of $r = \circ(x^2)$ using k-bit integer arithmetic and logic. We assume x is given by its standard encoding into an unsigned k-bit integer X. Since the exact result x^2 satisfies (4.10a), by Fact 2.1 the standard integer encoding R of its rounded value $\circ(x^2)$ is given by

$$R = D \cdot 2^{p-1} + L + b, \tag{4.11}$$

where

$$D = d + e_{\max} - 1, \tag{4.12}$$

$$L = \left\lfloor \ell \cdot 2^{p-1} \right\rfloor, \tag{4.13}$$

and b is the round bit defined in (2.10d).

Consequently, computing R amounts to deducing D, L, b from X, which we detail now in a parameterized fashion, that is, for the binary$k$ format. We illustrate our analysis for the binary32 format, and the corresponding C code (for rounding 'to nearest even') appears in Listing 4.1. As already said in Chapter 3, in order to give an idea of the ILP exposed by our approach, this C code has been set out in such a way that line i displays only the expressions that can be evaluated in $i+1$ cycles with the ST231 latencies.

Listing 4.1: Computing $\circ(x^2)$ for the binary32 format, $\circ = \mathrm{RN}$, and x generic.

```
0                                                          T2 = X & 0xff;
1  M = (X << 8) | 0x80000000;     E2 = (X >> 22) & 0x1fe;
2                                 F = 128 - E2;            c = M > 0xb504f333;
3                                 mu = max(c, F);
4  H = mul(M, M);
5  L = H >> (mu + 7);             G = H >> (mu + 6);       T1 = H << (26 - mu);
6
7
8  b =  (G & 1) && ((L & 1) | (T1 | T2));
9  return (((mu - F) << 23) + L) + b;
```

Computing L. From (4.10b) and (4.13) it follows that

$$L = \left\lfloor m^2 / 2^{1-p+\mu} \right\rfloor \tag{4.14}$$

and, therefore, we first need to deduce from X an integer encoding of m, say M, as well as the integer μ.

To produce M, recall that $m = (1.m_1 \ldots m_{p-1})_2$ as x is normal. Since $p \leqslant k$ a possible choice is to set up $m \cdot 2^{k-1}$, which can be obtained from (2.4) by shifting and masking:

$$M = m \cdot 2^{k-1} = (X \ll w) \mid 2^{k-1}.$$

For the binary32 format, where $w = 8$ and $k = 32$, this corresponds to line 1 in Listing 4.1; on ST231, this takes 2 cycles and, due to the extended immediate value $2^{31} = (80000000)_{16}$, 3 instruction syllables.

To get μ, recall first that x normal implies $e = E - e_{\max}$. Then, recalling that $e_{\min} = 1 - e_{\max}$ and applying (4.9),

$$\mu = \max(c, F), \qquad F = e_{\max} + 1 - 2E. \tag{4.15}$$

To get the boolean value c, it suffices to remark that (4.6) and $M = m \cdot 2^{k-1}$ imply

$$c = [M > M_0], \quad \text{with} \quad M_0 = \left\lfloor \sqrt{2} \cdot 2^{k-1} \right\rfloor.$$

To get the (possibly negative) integer F, first we extract $2E$ from X in (2.4) by using the identity

$$2E = \big(X \gg (p-2)\big) \mathbin{\&} (2^{w+1} - 2), \tag{4.16}$$

and then we subtract $2E$ from the constant $e_{\max} + 1$. For the binary32 format the computation of c and F appears at line 2 of Listing 4.1, where $(b504f333)_{16}$ is M_0 for $k = 32$; on ST231 each of c and F takes 3 cycles, so that μ is eventually obtained in 4 cycles.

Let us now see how to deduce L from M and μ. The property below shows that the k most significant bits of the $2k$-bit integer M^2 are enough for that purpose.

Property 4.6. *$L = \lfloor H/2^{\mu+w-1} \rfloor$ with $H = \lfloor M^2/2^k \rfloor$.*

Proof. From (4.14), $M = m \cdot 2^{k-1}$, and $k = w + p$ it follows that $L = \lfloor y/n \rfloor$ with $y = M^2/2^k$ and $n = 2^{\mu+w-1}$. Since $w \geqslant 3$ and since, by Property 4.5, $\mu \geqslant 0$, n is a positive integer. To conclude it suffices to apply the fact that $\lfloor y/n \rfloor = \lfloor \lfloor y \rfloor / n \rfloor$ for $n > 0$ (see [GKP94, p. 72]). □

Let `mul` denote a function that computes the higher half H of M^2. Then, by Property 4.6, we have:

$$H = \mathtt{mul}(M, M)$$
$$L = H \gg (\mu + w - 1)$$

For the binary32 format, this appears at lines 4 and 5 of Listing 4.1, and on the ST231, the `mul` function is implemented by calling the `mul64h` instruction. Since here $p = 24$ is even, we deduce from Property 4.5 that $\mu + 7$ is at most 31, which thus agrees with the C99 specification of the bitwise shift operator [Int99, p. 84]. With the ST231 latency constraints, both H and $\mu + 7$ are computed from X in 5 cycles, so that L is obtained in 6 cycles.

Computing b. We focus here on the most difficult case, rounding to nearest even, for which $b = g \wedge (\ell_{p-1} \vee t)$ with g and t as in (2.11).

Note first that g is the least significant bit of the integer $G = \lfloor \ell \cdot 2^p \rfloor = \sum_{0 \leqslant i \leqslant p} \ell_i 2^{p-i}$ and, using a proof similar to that of Property 4.6, we arrive at

$$g = G \bmod 2, \qquad G = \lfloor H/2^{\mu+w-2} \rfloor.$$

For the binary32 format, the corresponding C code appears at lines 5 and 8 of Listing 4.1. On ST231, G will have the same latency as L (6 cycles), and we thus get g in 7 cycles.

Since ℓ_{p-1} is the least significant bit of L we have

$$\ell_{p-1} = L \bmod 2,$$

so that it remains to compute the sticky bit t. The next result shows how to recover this bit simply by checking that some lower parts of H and X are nonzero, that is, without computing the lower half of the exact square M^2.

Property 4.7. *One has $t = [T_1 \neq 0] \vee [T_2 \neq 0]$ with T_1 and T_2 the two k-bit integers given by*

$$T_1 = H \ll (p + 2 - \mu) \quad \textit{and} \quad T_2 = X \bmod 2^{p - \lfloor k/2 \rfloor}.$$

Proof. Let q be the number of trailing zeros of $m = (1.m_1 \dots m_{p-1})_2$. Then m^2 can be written

$$m^2 = (s_{-1}s_0.s_1 \dots s_{2p-2q-2})_2 \quad \text{with} \quad s_{2p-2q-2} = 1.$$

Thus, $H = (s_{-1}s_0 \dots s_{k-2})_2$ and, using (4.10b) and (2.11), we can also decompose the sticky bit as $t = t_1 \vee t_2$ with

$$t_1 = s_{p-\mu+1} \vee \cdots \vee s_{k-2} \quad \text{and} \quad t_2 = s_{k-1} \vee \cdots \vee s_{2p-2q-2}.$$

By Property 4.5 and since $w \geqslant 3$, we have $k - 2 - (p - \mu + 1) + 1 = \mu + w - 2 \in \{1, \dots, k-1\}$. Hence $t_1 = 1$ if and only if the last $\mu + w - 2$ bits of H are not all zero, that is, if and only if the integer T_1 obtained by shifting H left by $k - (\mu + w - 2) = p + 2 - \mu$ is nonzero. Since $s_{2p-2q-2} = 1$ we have $t_2 = 0$ if and only if $k - 1 > 2p - 2q - 2$, that is, if and only if the number q of trailing zeros of m is at least $p - \lfloor k/2 \rfloor$. The latter condition is equivalent to $X \bmod 2^{p - \lfloor k/2 \rfloor} = 0$, which concludes the proof. □

For the binary32 format, Property 4.7 gives

$$T_2 = X \bmod 2^8,$$

which can be implemented by masking X as shown at line 0 of Listing 4.1. The computation of T_1 is a mere left shift of H by $26 - \mu$, the latter value ranging in $[0, 31]$ thanks to Property 4.5. Then notice that the bit $\ell_{p-1} \vee t$ is zero if and only if the integer U obtained by bitwise-ORing the integers L & 1 and T_1 and T_2 is zero. The logical AND of $g = G$ & $1 \in \{0, 1\}$ and U is thus enough to yield b, which allows us to avoid testing explicitly if T_1 or T_2 is nonzero. This is shown at line 8 of Listing 4.1. The parenthesization chosen there aims to reduce the overall latency for b on ST231: both L and T_1 can be obtained in 6 cycles, while T_2 costs 1 cycle; therefore, both L & 1 and $T_1 \,|\, T_2$ follow in 7 cycles, which yields b in 9 cycles.

Computing D. From the definitions of d and D in (4.10b) and (4.12) we deduce that $D = \mu + 2e + e_{\max} - 1$. Hence, recalling that $e = E - e_{\max}$ and the definition of F in (4.15),

$$D = \mu - F.$$

For the binary32 format, this subtraction appears at line 9 of Listing 4.1. Recalling that on ST231 we get F and μ in, respectively, 3 and 4 cycles, we will thus get D in 5 cycles.

Packing the result. From (4.11) the integer encoding R of the result satisfies $R = D \cdot 2^{p-1} + L + b$ and we have just detailed how to get from X the integers D, L, and b. Moreover, assuming the latency model of the ST231, their respective cost has been shown to be of 5, 6, and 9 cycles. This implies a latency of 6 cycles for $D \cdot 2^{p-1}$, and using the parenthesization shown at the last line of Listing 4.1 we eventually get R in 10 cycles. Thus, when $\circ$ is RN the overall cost is larger than that of the round bit b by only one cycle.

Some simplifications when $\circ$ is not RN. When the rounding mode $\circ$ is RD the round bit b in (2.10d) is zero. Consequently, the instructions involving G, T_1, T_2, and b can be suppressed and the last line of Listing 4.1 replaced with:

```
  return ((mu - F) << 23) + L;
```

When $\circ$ is RU the bit ℓ_{p-1} is not needed and b is the logical OR of $g = G \;\&\; 1$ and $T1 \,|\, T2$. In this case, we thus replace line 8 of Listing 4.1 by:

```
  b = (G & 1) || (T1 | T2);
```

With these new codes the expected latency of R on ST231 drops from 10 to 7 cycles for $\circ =$ RD, and from 10 to 9 cycles for $\circ =$ RU.

4.4 Detecting and handling special input

We first have to decide whether input x is special or not, that is, to compute from X the value of C_{spec} in (3.1). By Definition 4.1 this condition satisfies

$$C_{\text{spec}} = C_{\text{small}} \vee C_{\text{large}} \vee C_{\text{nan}} \tag{4.17}$$

with

- $C_{\text{small}} = \big[|x| < \alpha'\big]$,
- $C_{\text{large}} = \big[|x| \geqslant \Omega'\big]$,
- $C_{\text{nan}} = \big[x \text{ is NaN}\big]$.

The next two properties show how to obtain C_{small} and $C_{\text{large}} \vee C_{\text{nan}}$ by reusing the value $2E$ computed for the generic case (see (4.16) and line 1 of Listing 4.1).

Property 4.8. $C_{\text{small}} = [2E \leqslant e_{max} - p - 1]$.

Proof. Let $|X|$ and A' denote the standard integer encodings of $|x|$ and α', respectively. It is known [MBdD+10, p. 58] that $|x| < \alpha'$ if and only if $|X| < A'$, that is,

$$|X| \leqslant A' - 1. \tag{4.18}$$

By (2.4), $|X| = (E+\epsilon)\cdot 2^{p-1}$ with $\epsilon \in [0, 1-2^{1-p}]$ and, by Property 4.1, $A' = E'\cdot 2^{p-1}$ with $E' = \lfloor (e_{\min} - p)/2 \rfloor + e_{\max}$. Thus, (4.18) is equivalent to $E \leqslant E' - (\epsilon + 2^{1-p})$. Since E, E' are integers and $\epsilon + 2^{1-p} \in (0, 1]$, the latter inequality is equivalent to $E \leqslant E' - 1$. Now, $e_{\min} = 1 - e_{\max}$ gives $E' - 1 = \lfloor (e_{\max} - p - 1)/2 \rfloor$ and we conclude using the fact that $i \leqslant \lfloor j/2 \rfloor$ is equivalent, for integers i, j, to $2i \leqslant j$. □

Property 4.9. $C_{\text{large}} \vee C_{\text{nan}} = [2E \geqslant 3e_{max} + 1]$.

Proof. We use the same notation as in the proof of Property 4.8 and write O' for the standard integer encoding of Ω'. The special integer encoding used for NaNs gives $C_{\text{large}} \vee C_{\text{nan}} = [|X| \geqslant O']$. By Property 4.1 we have $O' = (3e_{\max}+1)/2 \cdot 2^{p-1}$, so that $|X| \geqslant O'$ is equivalent to $E + \epsilon \geqslant (3e_{\max} + 1)/2$. Since $\epsilon \in [0, 1)$, this is equivalent to $E \geqslant (3e_{\max} + 1)/2$ and the conclusion follows. □

For the binary32 format, a C fragment implementing C_{spec} by means of $2E$ and the two previous properties is shown at lines 1 to 3 of Listing 4.2. On ST231 the cost will be of 4 cycles and, as C_{spec} is independent of $\circ$, this fragment holds not only for $\circ = \text{RN}$ but also for $\circ \in \{\text{RD}, \text{RU}\}$.

Listing 4.2: Detecting and handling special input for the binary32 format and $\circ =$ RN.

```
 0                              absX = X & 0x7fffffff;
 1  E2 = (X >> 22) & 0x1fe; Cnan = absX > 0x7f800000;
 2  Csmall = E2 <= 102;    Clarge_or_nan = E2 >= 382;
 3  Cspec = Csmall || Clarge_or_nan;
 4  if (Cspec) {
 5     if (Csmall) return 0;                // r = +0
 6     else {
 7        if (Cnan) return 0x7fc00000;    // r = qNaN
 8        else return 0x7f800000; }        // r = +oo
 9  } else {
10    // generic case (Listing 1).
11  }
```

Once special input have been filtered out, it remains to return, for the given rounding mode $\circ$, the standard integer encoding R of the associated result r prescribed by (4.4):

When $\circ$ is RN. We deduce from (4.2) and (4.4) that for x special, r must be $+0$ if $|x| < \alpha'$, qNaN if x is NaN, and $+\infty$ otherwise. Implementing this is then straightforward as it suffices to recall from Table 2.2 that, on the one hand 0, $2^{k-1} -$

2^{p-1}, and $2^{k-1} - 2^{p-2}$ are standard encodings of $+0$, $+\infty$, and qNaNs, and that, on the other hand,

$$C_{\text{nan}} = \left[X \,\&\, (2^{k-1} - 1) > 2^{k-1} - 2^{p-1}\right].$$

For the binary32 format, the computation of C_{nan} is shown at lines 0 and 1 of Listing 4.2, while lines 5 to 8 display the computation of R. On ST231, lines 5 to 8 will be if-converted as shown by the following pseudo-code:

```
Rlarge_or_nan = slct(Cnan,0x7fc00000,0x7f800000)
R = slct(Csmall,0,Rlarge_or_nan)
```

With a latency of 1 cycle for the 'slct' instruction and since C_{nan} costs 2 cycles, we thus get R for x special in 4 cycles.

When $\circ$ is not RN. For $\circ =$ RD the only difference with the previous case is when $|x| \geqslant \Omega'$ and, by(4.2) and (4.4), we now have $r = \max(|x|, \Omega)$. The standard integer encoding of Ω is $2^{k-1} - 2^{p-1} - 1$, which equals $(7f7fffff)_{16}$ for the binary32 format. Consequently, it suffices to replace line 8 of Listing 4.2 with:

```
      else return maxu(absX, 0x7f7fffff);
```

For $\circ =$ RU, the specification differs from that for $\circ =$ RN only in the case where $|x| < \alpha'$, for which we have $r = \min(|x|, \alpha)$. Since the standard integer encoding of α is 1, an implementation for the binary32 format follows by simply replacing line 5 in Listing 4.2 by

```
   if (Csmall) return minu(absX, 1);
```

On ST231, R still costs 4 cycles as both $\max(|x|, \Omega)$ and $\min(|x|, \alpha)$ have a latency of 2 cycles, like C_{nan}.

4.5 Experimental results obtained on the ST231

The C codes detailed in Sections 4.3 and 4.4 yield a full implementation of squaring, for the binary32 format and each rounding mode. To check correctness we compiled them with gcc (using a C emulation of the `mul`, `max`, `maxu`, and `minu` operators as given in Appendix A), and compared with the results of multiplication $x \times x$ obtained on an Intel® Xeon® workstation. For each rounding mode this exhaustive test took about five minutes.

We also compiled our C codes with the ST200 compiler, in -O3 for the ST231 processor. The remainder of this section details the performances obtained in this context.

4.5.1 Operator performances

Latency and comparison with general multiplication. The latency on ST231 of our binary32 square implementation is shown in the third column of Table 4.1. Due to if-conversion, it gives for each rounding mode a number of clock cycles independent of the input value x. The values within square brackets indicate the lowest latencies we can theoretically achieve with the ST231 latency constraints and assuming unbounded parallelism; these best latencies follow from our analysis in Sections 4.3 and 4.4 and have the form $1+\mathcal{L}$ with $\mathcal{L}$ the best latency for the generic case. This first experiment shows that the latencies achieved in practice are at most 1 cycle form the best possible ones.

For comparison, the second column of Table 4.1 displays the latencies of the multiply operator $x \times y$ available in the FLIP 1.0 library and optimized for the ST231 [JR09a]. As shown in the fourth column, our specialization of this multiply operator into a square operator yields a speedup between 1.75 and 2.3, depending on the rounding mode.

$\circ$	FLIP 1.0 multiply	square	speedup
RN	21	12 [11]	1.75
RD	21	9 [8]	2.3
RU	21	11 [10]	1.9
RZ	18	9 [8]	2

Table 4.1: Latency comparison for square and multiply.

Instruction-level parallelism. When designing our algorithms in Sections 4.3 and 4.4 we strived to expose as much ILP as we could. As already mentioned in the introduction of this thesis, to evaluate ILP in practice we use instructions-per-cycle (IPC), which is the parallelism really exposed on the target. As shown in Table 4.2 it is deduced from the assembly code by dividing the number of instructions by the latency. The IPC achieved is close to the highest ILP reachable within the architectural constraints of the machine, demonstrating a very efficient usage of its resources.

$\circ$	Latency L	Number N of instructions	IPC = N/L
RN	12	42	3.5
RD	9	31	3.4
RU	11	37	3.4

Table 4.2: Latency, code size, and IPC for square.

Comparison with two non-IEEE variants. To study the impact on latency of relaxing the IEEE 754 specification used so far, we have implemented for each rounding mode a finite-math-only variant and a variant without subnormals.

Finite math only. We assume here that input and output are neither infinity nor NaN, and that overflow does not occur. Hence x now satisfies $|x| < \Omega'$. On the one hand, this leaves the generic path unchanged, so that the best possible latencies are the same as for our IEEE version. On the other hand, (4.17) becomes $C_{\text{spec}} = C_{\text{small}}$ and the C codes of Section 4.4 can be simplified accordingly. As the third column of Table 4.3 shows, in practice this simplification has no impact on latency for RD, while it saves 1 cycle for RN and RU.

No subnormals. Here we assume that x is not subnormal, which means, writing $\lambda = 2^{e_{\min}}$ for the smallest positive normal number, that x is either NaN, zero, or such that $\lambda \leqslant |x|$. We also assume that if the exact result x^2 lies in the subnormal range $(0, \lambda)$ then $r = +0$ for RN and RD, and $r = \lambda$ for RU. Since $x^2 < \lambda$ is equivalent to $|x| < \lambda'$ with $\lambda' = \sqrt{\lambda}$, the specification of this non-IEEE variant can thus be deduced from (4.2) and (4.4) simply by replacing α and α' with, respectively, λ and λ'.

For the special path, this relaxed specification implies $C_{\text{small}} = [|x| < \lambda']$ and the proof of Property 4.8 can be adapted to show that we now have $C_{\text{small}} = [2E \leqslant e_{\max} - 1]$. Thus, it suffices to replace 102 by 126 at line 2 of Listing 4.2 and, for RU, to replace 1 by 2^{23} in the `minu` operation. These updates clearly have no impact on the latency.

Concerning the generic path, the case (4.8b) need not be considered anymore, so that μ in (4.9) is now equal to c. Hence the `max` operation at line 3 of Listing 4.1 can be removed and we can replace μ by c at line 5. However, as H still has a latency of 5 cycles, this simplification does not shorten the critical path. This means that the best latencies that we can theoretically achieve with this second non-IEEE variant are the same as for our IEEE version. Furthermore, Table 4.3 shows that this is true in practice as well.

$\circ$	IEEE	finite math only	no subnormals
RN	12	11	12
RD	9	9	9
RU	11	10	11

Table 4.3: Latency comparison with two non-IEEE variants.

4.5.2 Application examples

We now apply our fast operator to some square-intensive algorithms in order to study its effect on real applications. After giving a theoretical model of the speedup achievable on ST231, we show practical speedups and how they match that model. All experiments have been done on the ST231 cycle-accurate simulator, and while we focus on rounding 'to nearest even' (RN) similar results hold for RU and RD.

Speedup model for loop nests involving squares. While Table 4.1 gives speedups for a single replacement of multiply by square, we now evaluate the theoretical expectation of speedup for loops. When the square operator appears in a loop of n iterations, we introduce the definition

$$\text{theoretical speedup} = \frac{(\mathcal{L} + \Delta \cdot \sigma) \cdot n + \mathcal{C}}{\mathcal{L} \cdot n + \mathcal{C}}$$

with $\mathcal{L}$, Δ, σ, and $\mathcal{C}$ given as follows:

- $\mathcal{L}$ is the latency of the loop body where squares are used, which includes the application-related operations inside the loop and the cost to control the loop.

- Δ is the latency gap between multiply and square.
- σ is the number of squares in a single iteration.
- $\mathcal{C}$ is the cost of the straight-line code outside the loop.

Note that when n tends to infinity, the theoretical speedup tends to $1 + \Delta \cdot \sigma / \mathcal{L}$, which does not depend on $\mathcal{C}$.

On ST231 the values Δ, $\mathcal{L}$, and $\mathcal{C}$ have the following features. First, Table 4.1 gives $\Delta = 21 - 12 = 9$ cycles. Second, $\mathcal{L}$ and $\mathcal{C}$ can be modelled as

$$\mathcal{L} = \mathtt{a} \cdot \mathtt{Br} + \mathtt{Idx} + \mathtt{Ld} + \mathtt{C}_{\mathtt{in}}, \qquad \mathcal{C} = \mathtt{St} + \mathtt{C}_{\mathtt{out}}.$$

Here, `Br` is the cost of a taken branch, which is 3 cycles. The unrolling transformation done by the compiler has the effect of dividing the number of branches taken to jump back to the head of the loop by the unrolling factor; `a` in $(0, 1]$ is used to denote this effect. The value of `a` depends on the application and on the compiler optimization level; to estimate the trend of the speedup, we take $\mathtt{a} = 0.5$, meaning that the loop is always unrolled by a factor of 2. `Idx` is the cost to test loop index, which is 1 cycle; `Ld` is the cost to load input values, which is 3 cycles. $\mathtt{C}_{\mathtt{in}}$ (resp. $\mathtt{C}_{\mathtt{out}}$) is the latency of the application-related code within (resp. outside) the loop; for each application, the values of $\mathtt{C}_{\mathtt{in}}$ and $\mathtt{C}_{\mathtt{out}}$ can be deduced from the latencies of the 5 basic operators of FLIP 1.0 [Rev09, Table 1] and from the latency of 12 cycles of our square operator. Finally, `St` is the cost of stack handling of the function call; it ranges from 10 to 12 cycles depending on the achievable ILP of each application.

The above model has been applied to three application examples, which we review now.

Example 1: Euclidean norm. Given a vector v of n floating-point data v_i we consider the computation of its Euclidean norm

$$\|v\|_2 = \left(\sum_{i=1}^{n} v_i^2 \right)^{1/2}$$

by means of three different algorithms.

Naive algorithm. Here $\|v\|_2$ is produced by a for loop whose ith iteration simply squares v_i and adds it to the current partial sum of squares. Figure 4.1 shows that the theoretical speedup tends to about 1.185 and that in practice we are close to this value as soon as $n \geqslant 20$, assuming n is known at runtime. If n is known at compile time then an even higher speedup is observed, since the compiler achieves more efficient loop unrolling optimization. The nine black bullets in Figure 4.1 illustrate this fact for $n = 2, \ldots, 10$. The greatest speedups are for $n \leqslant 4$, since in this case all the parameters are directly passed to the function by registers instead of by stack and the loop is fully unrolled.

Two-pass algorithm. This algorithm, which is already mentioned in [Blu78], aims to avoid overflow by first computing $\|v\|_\infty = \max_i |v_i|$ and then applying the naive algorithm to the scaled vector $[v_i / \|v\|_\infty]_i$. The input is scanned twice and n divisions are used. Thus, the speedup we can expect is lower than for the naive algorithm, as

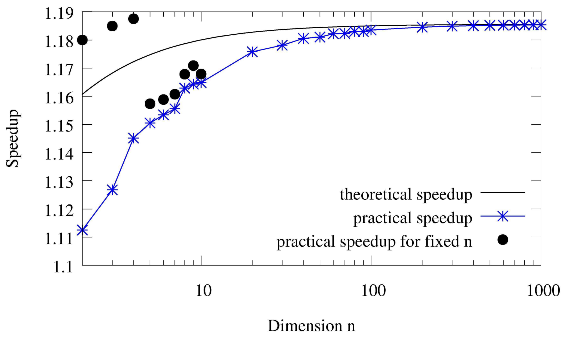


Figure 4.1: Impact of square on naive Euclidean norm.

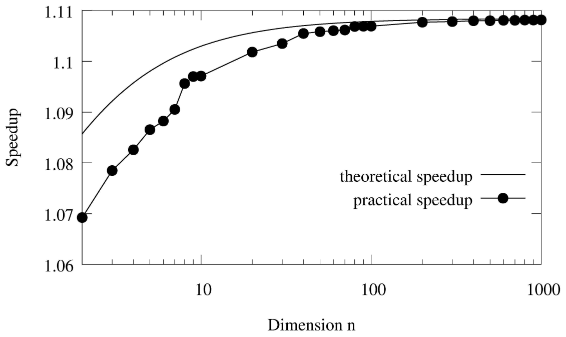


Figure 4.2: Impact of square on two-pass Euclidean norm.

shown in Figure 4.2. However, we see that the practical speedup still matches well the theoretical model as soon as $n \geqslant 20$.

One-pass algorithm. This algorithm also intends to avoid overflow but requires only one pass over the data, the scaling factor being now computed on the fly. It is an adaptation of Blue's algorithm [Blu78] attributed to Hammarling and implemented in LAPACK [Hig02, p. 507]. Each of the n iterations performs exactly 1 square as well as 2 or 4 additional operations, depending on the data v_i. The total number of operations thus varies dynamically and turns out to be maximum when $|v_1| < |v_2| < \cdots < |v_n|$, and minimum when $|v_1| \geqslant |v_2| \geqslant \cdots \geqslant |v_n|$. Each of these two extreme cases yields a behavior similar to the one displayed in Figure 4.2, the limiting value when $n \to \infty$ being about 1.077 in the first case, and about 1.096 in the second case.

Example 2: sample variance. The sample variance of $v_1, \ldots, v_n$ is

$$\frac{1}{n-1}\sum_{i=1}^{n}(v_i - \bar{v})^2, \qquad \text{where} \qquad \bar{v} = \frac{1}{n}\sum_{i=1}^{n} v_i.$$

It is known [Hig02, p. 11] that it can be evaluated accurately by the naive way, which requires two passes over the data: get $\bar{v}$ first and then iteratively square and add the $v_i - \bar{v}$'s. This method has the same structure as the two-pass algorithm in the previous example (with the maximum replaced by a sum, and the division replaced by a subtraction). Consequently, the theoretical and practical speedups brought by our fast square operator are similar to those in Figure 4.2.

Example 3: binary powering. For a floating-point datum x and for n an integer power of two such that $n \geqslant 4$, we consider here the evaluation of x^n by means of $\log_2 n$ successive squares. The results obtained for this application are shown in Figure 4.3. The qualitative analysis done for the naive algorithm of Example 1 still applies but, since binary powering does not involve any operation other than squaring, higher speedups are observed. This is even more visible for small values of n (say, between 4 and 64) for which loop unrolling is done by the compiler.

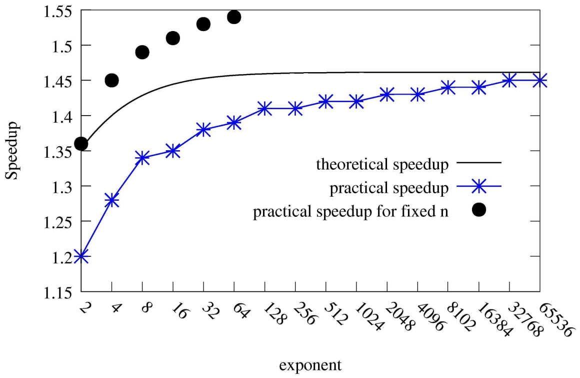


Figure 4.3: Impact of square on binary powering.

Chapter 5

Scaling by integer powers of two

Here we consider floating-point scaling, with subnormal support and correct rounding for all standard specified modes. By scaling, we mean multiplication by an integer power of two: given a floating-point datum x and an integer n, we want $x \cdot 2^n$. Depending on the sign of n, either overflow or inexact result may happen, which leads to very different algorithms. Therefore, we first separate the discussion for nonnegative and negative n, and then we propose a complete implementation by simply merging the two cases. On a VLIW processor like the ST231, this method results in a low latency and high ILP scaling operator. Moreover, we present very efficient implementations for some specific values of n, such as 1 and -1, which indeed compute $2x$ and $x/2$ and can be considered as special cases of multiplication. The numerical results show that it is worthwhile to have dedicated algorithms for different values of n and that this basic operator can even impact some high-level applications like matrix balancing.

5.1 Introduction

In this chapter we deal with the scaling operation, which given a floating-point datum x and an integer n, evaluates $x \cdot 2^n$. The main advantage of having such an operation is that it allows to multiply by 2^n without having to compute explicitly this value. Multiplication by such integer powers of two is useful in various circumstances, such as the following ones:

- **Multiplication and division by small constants:** in many C codes, statements like `*2.0f`, `*4.0f`, `*0.5f`, `*0.25f` can be seen, which correspond to very special cases of scaling. Furthermore, codes for computing inverse FFTs in dimension $N = 2^n$ typically end with the division by N of each of the computed values, and in practice N is often known at compile time (for example, $N = 256$) [PTVF07, §12].

- **Control of underflow and overflow:** scaling by integer powers of two is also sometimes used to improve the behavior of numerical algorithms, especially in order to avoid or reduce the occurrence of overflow and unnecessary underflow. For example, such scalings have been introduced in Horner's method for polynomial evaluation [HPW90] and in algorithms for performing divisions of complex floating-point numbers [Pri04].

- **Improvement of numerical accuracy:** finally, scaling is also used to increase the accuracy of eigenvalue computations. In this context, a common preliminary step is to 'balance' the input matrix A by means of diagonals of powers of two: A is transformed into DAD^{-1} and D is chosen so as to make the norms of the rows and columns of A of the same order of magnitude. The balanced matrix DAD^{-1} has the same eigenvalues of A but, in most cases, is easier to handle by eigenvalue solvers [PR69], [Bet08], [PTVF07, §11.6.1]. This balancing procedure is proceeded by default in MATLAB's `eig` function [HH05].

Scaling was already recommended in the 1985 version of the IEEE 754 standard. In the 2008 revision, it is now required and its prototype is as follows [IEE08, §5.3.3]:

$$\textit{sourceFormat}\ \mathrm{scaleB}(\textit{source},\ \textit{logBFormat}),$$

where *sourceFormat* denotes the destination format, *source* is the format of floating-point input x, and *logBFormat* is the format used for n. In particular, *logBFormat* can be either a floating-point format or an integer format, and it must have enough range to include all the integers from $-2(e_{\max} + p)$ to $2(e_{\max} + p)$.

Also in §5.3.3 of the standard, another function using *logBFormat* is the logB function and its prototype is

$$\textit{logBFormat}\ \mathrm{logB}(\textit{source}).$$

This function logB returns the exponent of its input in the return format.

For any floating-point number x and *logBFormat* scaling exponent n, the scaling function $(x, n) \mapsto x \cdot 2^n$ is fully specified by the standard as follows, assuming $\circ$ is one of the standard rounding modes.

$$r = \begin{cases} x & \text{if } x = \pm\infty, \\ \mathrm{qNaN} & \text{if } x \text{ is NaN}, \\ \circ(x \cdot 2^n) & \text{otherwise.} \end{cases} \tag{5.1a}$$

Furthermore, in all our designs, we assume that

$$n \text{ is a signed 32-bit integer.} \tag{5.1b}$$

This means that we choose *signed 32-bit integer* format for *logBFormat*. There are two reasons for this design decision. First, the general purpose registers on the ST231 are 32-bit wide. Second, signed 32-bit integers are enough for the binary32, binary64, and binary128 formats: indeed, as shown in the table below, in all these cases $2(e_{\max} + p)$ is less than $2^{31} = 2147483648$.

	$e_{\max}$	p	$2(e_{\max} + p)$
binary32	127	24	302
binary64	1023	53	2152
binary128	16383	113	32992

Remark that since we are assuming radix 2 floating-point arithmetic, the scaleB function is equivalent to `scalbn` for radix 2 or `ldexp` in C11 [Int11].

There are a number of implementations for scaling, which assume that basic floating-point arithmetic support is available. For example, Cody and Coonen's algorithm [CC93] is purely based on floating-point arithmetic, which can result in good portability between different floating-point systems. FDLIBM [fdl] uses a hybrid method which uses fixed-point arithmetic to handle normal numbers and floating-point arithmetic for subnormal numbers. However, as we focus on integer-only processors, all these solutions cannot really be efficient on our target.

Depending on the sign of n, either overflow or inexact result may happen, which leads to very different algorithms. Therefore, we first separate the discussion for non-negative and negative n, and then we propose a complete implementation by simply merging the two cases. On a VLIW processor, this method results in a low latency and high ILP scaling operator.

Moreover, since scaleB can be used as a special case of floating-point multiplication and division by constant when n is known at compilation time, we present very efficient implementations for some specific values of n, such as 1 or -1, which correspond to, respectively, multiplication or division by two.

Outline. This chapter is organized as follows. The cases $n \geqslant 0$ and $n < 0$ are presented in §§ 5.2 and 5.3, respectively. In each case, we provide: a specification, an algorithm, some implementation details, a specialized version for small values of $|n|$, and performance results. Then, §5.4 describes our general scaleB operator and its application to matrix balancing.

5.2 Scaling by nonnegative powers of two

Throughout this section, we assume

$$n \geqslant 0.$$

This case is the simplest one in the sense that there is no rounding to perform. Consequently, the main difficulty here is to support subnormal numbers efficiently.

5.2.1 Specification

As explained in Section 3.2.1, the classification of *generic* and *special* inputs is essential to achieve high ILP. For this, we start by giving a property characterizing overflow.

Property 5.1. *Let x be a finite floating-point number and let n be a nonnegative integer. Then*

- *overflow occurs if and only if $|x| \geqslant \Omega'$ with*

$$\Omega' := 2^{e_{max}+1-n};$$

- *when overflow does not occur, we have $\circ(x \cdot 2^n) = x \cdot 2^n$.*

Proof. Let us show the first claim. Since x is finite, we have $|x| = m_x \cdot 2^{e_x}$ with $m_x = (m_{x,0}.m_{x,1} \dots m_{x,p-1})_2$. Hence, $|x|2^n = m_x \cdot 2^{e_x+n}$ still has at most $p-1$ fraction bits, so that overflow occurs if and only if $|x|2^n \geqslant 2^{e_{\max}+1}$, that is, if and only if $|x| \geqslant \Omega'$, as wanted.

For the second claim, note that the absence of overflow implies $|x|2^n < 2^{e_{\max}+1}$. But since x is a radix-2 floating-point number, this strict inequality implies $x \cdot 2^n$ is a finite floating-point number, so that $\circ(x \cdot 2^n) = x \cdot 2^n$. □

Thanks to the above property and by using the rules in §2.3, we can now refine the specification given in (5.1). This is shown in Table 5.1 and leads to the following definition.

$\circ(x \cdot 2^n)$	x					
	finite				$\pm\infty$	NaN
	$\lvert x\rvert \in (0, \Omega')$	$x \leqslant -\Omega'$	$x \geqslant \Omega'$	$x = \pm 0$		
RN	$x \cdot 2^n$	$-\infty$	∞	x		qNaN
RU		$-\Omega$	∞			
RD		$-\infty$	Ω			
RZ		$-\Omega$	Ω			

Table 5.1: Refined IEEE specification of scaling when $n \geqslant 0$.

Definition 5.1. *For scaling and when $n \geqslant 0$, input x is* **generic** *when $|x| \in (0, \Omega')$, and* **special** *otherwise.*

Note that $x = \pm 0$ is considered as special in order to simplify the handling of generic input. Indeed, we will see that for scaling by nonnegative powers of two, the handling of generic input is on the critical path, and therefore we try to move some computation to the special path in order to balance the costs of the two paths to decrease the whole latency.

The gray part of Table 5.1 specifies the results for special inputs. More precisely, $C_{\text{spec}} := [x \text{ is special}]$ satisfies

$$C_{\text{spec}} = \big[x \text{ is } \pm 0, \pm\infty \text{ or NaN}\big] \vee \big[x \text{ is finite and } |x| \geqslant \Omega'\big]. \tag{5.2}$$

5.2.2 Scaling generic input

Assume that x is generic. According to Definition 5.1, this implies that both x and $x \cdot 2^n$ are nonzero, finite floating-point numbers. In other words, there is no rounding issue and all we have to do is to return this exact result efficiently even for subnormal inputs.

To achieve this goal, we start with the theorem below, which provides a formula for handling simultaneously normal and subnormal inputs. Recall that the k-bit unsigned integer X denotes the standard encoding (abbreviated as encoding) of floating-point number x for binary$k$ format. Recall also that $|X|$ stands for the encoding of the absolute value of x, that is $|X| = X \bmod 2^{k-1}$.

Theorem 5.1. *Let x be generic in the sense of Definition 5.1, let s_x denote the sign bit of x, let λ_x be the number of leading zeros of the significand of x, and let μ be defined as*

$$\mu = \min(\lambda_x, n). \tag{5.3a}$$

Then, the encoding R of the exact result $r = x \cdot 2^n$ can be deduced from the encoding X of x as follows:

$$R = s_x \cdot 2^{k-1} + |R| \tag{5.3b}$$

with

$$|R| = |X| \cdot 2^{\mu} + (n - \mu) \cdot 2^{p-1}. \tag{5.3c}$$

Proof. When x is generic, $r = (-1)^{s_x} \cdot m_x \cdot 2^{e_x+n}$, and $|r| = m_x \cdot 2^{e_x+n}$. Thus, $R = s_x \cdot 2^{k-1} + |R|$, as claimed in Equation (5.3b).

To prove Equation (5.3c), recall from Chapter 2 that

$$|X| = (e_x - e_{\min} + m_x) \cdot 2^{p-1}$$

and

$$m_x = (\underbrace{0.0\ldots0}_{\lambda_x \text{ zeros}}\,1 m_{x,\lambda_x+1} \ldots m_{x,p-1})_2.$$

When x is normal, $\lambda_x = 0$, and $|r| = |x| \cdot 2^n = m_x \cdot 2^{e_x+n}$ is a finite floating-point number with $m_x \in [1,2)$ and $e_x + n \in [e_{\min}, e_{\max}]$. Therefore, the encoding of r satisfies $|R| = (e_x - e_{\min} + n + m_x) \cdot 2^{p-1} = |X| + n \cdot 2^{p-1}$, which is equivalent to 5.3c for $\mu = \min(0, n) = 0$.

When x is subnormal, $e_x = e_{\min}$ and we distinguish between two sub-cases:

- When $n \leqslant \lambda_x$, $|r| = m_x \cdot 2^n \cdot 2^{e_{\min}}$ is a finite floating-point number with $m_r = m_x \cdot 2^n \in (0,2)$ and $e_r = e_{\min}$. Therefore, we have

  $$\begin{aligned} |R| &= (e_r - e_{\min} + m_r) \cdot 2^{p-1} \\ &= m_x \cdot 2^n \cdot 2^{p-1} \\ &= |X| \cdot 2^n, \end{aligned}$$

  which is equivalent to (5.3c) for $\mu = n$.

- When $n > \lambda_x$, we now have $m_r = m_x \cdot 2^{\lambda_x}$ and $e_r = e_{\min} + n - \lambda_x$. Hence

  $$\begin{aligned} |R| &= (e_r - e_{\min} + m_r) \cdot 2^{p-1} \\ &= m_x \cdot 2^{\lambda_x} \cdot 2^{p-1} + (n - \lambda_x) \cdot 2^{p-1} \\ &= |X| \cdot 2^{\lambda_x} + (n - \lambda_x) \cdot 2^{p-1}, \end{aligned}$$

  which is equivalent to (5.3c) for $\mu = \lambda_x$.

□

The ready-to-implement equations in Theorem 5.1 lead to the result `Rgen` of generic input whose implementation for the binary32 format is shown in Listing 5.1.

Listing 5.1: Implementation of R for *binary32* as in Theorem 5.1.

```
0 absX  = X & 0x7fffffff;
1 nz = clz(absX);        sigX = X-absX;
2
3 lambda = maxu(nz,8)-8;
4 mu = minu(lambda,n);
5 Xmu = absX << mu;      D = n - mu;
6
7 Rgen = (sigX | Xmu) + (D << 23);
```

Here, the hardware instruction `clz` is employed to compute the number of leading zeros of $|X|$. As can be seen from Listing 5.1, the latency for `nz` is of 2 cycles. Then, the comparison of `nz` with the length of the exponent field ($w = 8$ for binary32), is carried out by using `maxu`, so that the latency for λ_x (`lambda`) is of 4 cycles.

Note also that this code exposes some ILP: `nz` and `sigX` can be computed simultaneously (see line 1), and this holds for `sigX` $|$ `Xmu` and `D` $<<$ `23` as well (see line 7). Specifically, `sigX` $|$ `Xmu` and `D` $<<$ `23` have a latency of 7 cycles, so that the latency of `Rgen` is of 8 cycles

5.2.3 Detecting and handling special input

Detecting special input. This first step corresponds to the evaluation of C_{spec} and is independent of the rounding mode. According to Equation (5.2), C_{spec} can be implemented as

$$C_{\text{spec}} = C_{\text{zero}} \vee C_{\text{large}} \vee C_{\text{nan}},$$

where

$$C_{\text{zero}} = [x = \pm 0], \quad C_{\text{large}} = [|x| \geqslant \Omega'], \quad C_{\text{nan}} = [x \text{ is NaN}].$$

Recalling that $|X|$ is the encoding of $|x|$, a straightforward computation of C_{zero} is as follows

$$C_{\text{zero}} = [|X| = 0]. \tag{5.4}$$

On the other hand, the following property shows how to get $C_{\text{large}} \vee C_{\text{nan}}$ by reusing the variable `lambda` computed in the generic path (see line 3 in Listing 5.1).

Property 5.2. *We have* $\mathrm{C}_{large} \vee \mathrm{C}_{nan} = [E_x + n' - \nu - \Lambda \geqslant 2e_{max}]$ *with*

$$n' = \min(n, 2e_{max} + p - 1), \quad \nu = [\text{clz}\,|X| \leqslant w],$$

and

$$\Lambda = \max(\text{clz}\,|X|, w) - w.$$

Before proving the property, let us remark that during the implementation of $C_{\text{large}} \vee C_{\text{nan}}$, the reason for using n' instead of n is to avoid the overflow caused by the addition $E_x + n$.

Proof. First, let us prove that for any floating-point number x,

$$\circ(x \cdot 2^{n}) = \circ(x \cdot 2^{n'}). \tag{5.5}$$

When x is NaN or $\pm\infty$, Equation (5.5) holds according to the specification shown in Table 5.1. When x is finite and $n < 2e_{\max} + p - 1$, we have $n' = n$ and thus (5.5) is true. When x is finite and $n \geqslant 2e_{\max} + p - 1$, we have $\Omega' = 2^{e_{\max}+1-n} \leqslant \alpha$; since overflow occurs if and only if $|x| \geqslant \Omega'$ according to Property 5.1, and $|x| \geqslant \alpha$ for finite x, we are in a case where $\circ(x \cdot 2^{n})$ and $\circ(x \cdot 2^{n'})$ overflow, and thus are equal to each other.

By the definition of ν and Λ, we have

$$\nu = \begin{cases} m_{x,0}, & \text{if } x \text{ is a finite floating-point number,} \\ 1, & \text{if } x \text{ is } \pm\infty \text{ or NaN,} \end{cases}$$

and

$$\Lambda = \begin{cases} \lambda_x, & \text{if } x \text{ is a finite floating-point number,} \\ 0, & \text{if } x \text{ is } \pm\infty \text{ or NaN.} \end{cases}$$

Now, let us prove that when C_{large} or C_{nan} holds, $E_x + n' - \nu - \Lambda \geqslant 2e_{\max}$ holds. To do so, we distinguish between two cases:

- When x is $\pm\infty$ or NaN, we have $E_x = 2^{w} - 1$, $\nu = 1$, and $\Lambda = 0$. Since $n \geqslant 0$, we have also $n' \geqslant 0$, from which we deduce that $E_x + n' - \nu - \Lambda \geqslant 2e_{\max}$.
- When x is finite and overflows occurs, $|x| \cdot 2^{n'} = m_x \cdot 2^{\lambda_x} \cdot 2^{e_x - \lambda_x + n'} \geqslant 2^{e_{\max}+1}$. Since $m_x \cdot 2^{\lambda_x}$ is in $[1, 2)$, we have $2 \cdot 2^{e_x - \lambda_x + n'} > 2^{e_{\max}+1}$, that is, $e_x - \lambda_x + n' \geqslant e_{\max} + 1$. Since $E_x = e_x - e_{\min} + m_{x,0}$, $m_{x,0} = \nu$, and $\lambda_x = \Lambda$, we obtain $E_x + n' - \nu - \Lambda \geqslant 2e_{\max}$.

On the other hand, assume now that neither C_{large} nor C_{nan} holds. This implies that x is a finite floating-point number and that overflow does not occur, so that $E_x + n' - \nu - \Lambda < 2e_{\max}$. □

To summarize, by using Equation (5.4) and Property 5.2, we compute C_{spec} as

$$C_{\text{spec}} = C_{\text{zero}} \vee (C_{\text{large}} \vee C_{\text{nan}}).$$

The corresponding C code for binary32 is shown at lines 0 to 5 of Listing 5.2.

Handling special input. Once we have filtered out special input by C_{spec}, it remains to handle them according to the specification in Table 5.1. From this table, remark that this second step does depend on the rounding mode $\circ$.

Assume first that $\circ$ = RN. For this rounding mode, special input handling is done as shown at lines 7 to 10. Notice that we first handle NaN input by using an extra condition, namely

$$C_{\text{nan}} = |X| > \iota(+\infty), \qquad \iota(+\infty) = (2e_{\max} + 1) \cdot 2^{p-1}.$$

Then, it remains to return either x or $(-1)^{s_x} \cdot \infty$, depending on whether C_{zero} is true or not.

Listing 5.2: Detecting and handling special input of scaling when $n \geqslant 0$ for the binary32 format, $\circ = \text{RN}$.

```
 0 absX = X & 0x7fffffff; np = minu(n,277);
 1 nz = clz(absX); Ex = absX >> 23; Cnan = absX > 0x7F800000;
 2 nu = nz <= 8;   Czero = absX == 0;  sigX = X - absX;
 3 lambda = maxu(nz,8)-8;
 4 Clarge_or_nan = (int32_t)(Ex+np-nu-lambda) >= 2*0x7f;
 5 Cspec = Czero || Clarge_or_nan;
 6 if (Cspec) {
 7    if (Cnan) return 0x7fc00000;           // r = qNaN
 8    else {
 9       if (Czero) return X;                // r = +\- 0
10       else return sigX|0x7f800000; }      // r = +\- oo
11 } else {
12   // generic case (Listing 5.1).
13   return Rgen;
14 }
```

For other rounding modes (RU, RD, RZ), all we need to change is lines 9 and 10 according to Table 5.1. The corresponding C codes for the binary32 format are given below.

- Rounding up ($\circ = \text{RU}$):

```
if(Czero|(Ex == 255)) return X;
else return sigX|(0x7f800000- (sigX!=0));
```

- Rounding down ($\circ = \text{RD}$):

```
if(Czero|(Ex == 255)) return X;
else return sigX|(0x7f800000- (sigX==0));
```

- Rounding to zero ($\circ = \text{RZ}$):

```
if(Czero|(Ex == 255)) return X;
else return sigX|0x7f7fffff;
```

Since Listing 5.2 refers to Listing 5.1 at line 12, we have in fact here a complete code for scaling when $n \geqslant 0$. Consequently, we present the performances of this operator in the paragraph below.

Performances on the ST231. Table 5.2 summarizes the performances of scaling when $n \geqslant 0$ on the ST231. We give both the latencies in cycles and the number of instructions (in brackets) for each rounding mode.

For the binary32 format, we see from Listing 5.1 that it takes 8 cycles to compute the result if the input is generic. Then, ideally, it would cost only one more cycle to obtain the desired result, where a selection between the results computed by the generic path and by the special path is applied. In Table 5.2, for $\circ$ = RN, it takes exactly 9 cycles, which means that the handling of generic input is critical to the latency here and the code is optimally scheduled. For other rounding modes, although there is an overhead of two or four instructions, only one more cycle is required.

Although the algorithm is optimized for the binary32 format, we still achieve interesting latencies for the binary64 format: the data width is doubled while the latencies are less than twice those for binary32.

For both formats and all the rounding modes, the IPCs (instructions per cycle, see §4.5.1) are higher than 3, which means that for all these implementations most of the bundles are full.

$\circ(x \cdot 2^n), n \geqslant 0$	RN	RU	RD	RZ
binary32	9 [28]	10 [32]	10 [32]	10 [30]
binary64	14 [44]	16 [51]	16 [51]	15 [46]

Table 5.2: Latencies in # cycles [code sizes in # instructions] for scaling when $n \geqslant 0$.

5.2.4 Specializing to small values of n

Scaling by summation. We introduce here another algorithm to compute scaling, which does not require λ_x, the number of leading zeros of the significand of x. Instead, we use n times the min function and n additions to compute $x \cdot 2^n$ for x generic or zero. Assuming infinite parallelism, this method leads to a latency of $O(\log_2 n)$ by using parallel summation. In practice, this method turns out to be fast for small values of n on the ST231, and therefore it is used for some specialized cases of scaling, such as **mul2**. The theorem below gives the formula that underlies this algorithm.

Theorem 5.2. *Let x be generic or zero. Then, the encoding R of the exact result $r = x \cdot 2^n$ can be deduced from the encoding X of x as follows:*

$$R = X + \sum_{i=0}^{n-1} \min(|X| \cdot 2^i, 2^{p-1}). \tag{5.6}$$

Proof. When x is zero, we have $r = x$ and thus also $R = X$. Assume now that x is nonzero, which implies in particular

$$|X| \geqslant 1.$$

For $n = 0$ the result is clear, and for $n \geqslant 1$ we proceed by induction on n.

- If $n = 1$ then $r = 2x$. When x is normal, we have $|X| \geqslant 2^{p-1}$ so that $\min(|X|, 2^{p-1}) = 2^{p-1}$ and, by Theorem 5.1 we deduce that $\mu = 0$ and

$$\begin{aligned} R &= s_x \cdot 2^{k-1} + |X| + 2^{p-1} \\ &= X + \min(|X|, 2^{p-1}). \end{aligned}$$

When x is subnormal, we have $|X| < 2^{p-1}$, and Theorem 5.1 gives $\mu = 1$ and

$$\begin{aligned} R &= s_x \cdot 2^{k-1} + 2|X| \\ &= X + |X| \\ &= X + \min(|X|, 2^{p-1}). \end{aligned}$$

Therefore, we have shown the result for $n = 1$.

- Assume now that $n \geqslant 2$ and that the result is true up to $n-1$. Defining $y = 2x$, one has $r = y \cdot 2^{n-1}$ and one may check that since x is generic, y is generic as well. Thus, the induction assumption yields $R = Y + \sum_{i=0}^{n-2} S_i$, where Y is the standard encoding of y, and where $S_i = \min(2^i|Y|, 2^{p-1})$. Furthermore, the case $n = 1$ studied just before leads to

$$Y = X + \min(|X|, 2^{p-1}). \tag{5.7}$$

Consequently, it suffices to check that

$$S_i = \min(2^{i+1}|X|, 2^{p-1}) \quad \text{for } i = 0, \ldots, n-2. \tag{5.8}$$

If $|X| \leqslant 2^{p-1}$ then (5.7) gives $|Y| = 2|X|$, from which (5.8) follows. If $|X| > 2^{p-1}$ then $|Y| = |X| + 2^{p-1} \geqslant 2^{p-i-1}$, which implies $S_i = 2^{p-1}$; since in this case $|X| \geqslant 2^{p-i-2}$, one has (5.8) too. Combining (5.7) and (5.8) thus gives the desired expression for R, and the conclusion follows by induction.

□

Here are three examples of Theorem 5.2 for some specific values of n and the corresponding C codes for the binary32 format:

- Example where $n = 1$: we have $r = 2x$, which is the **mul2** operator, and thus

$$R = X + \min(|X|, 2^{p-1}).$$

Getting R from X takes 3 cycles on the ST231, as the listing below shows:

```
0 absX = X & 0x7FFFFFFF;
1
2 Rgen = X + min(absX, 0x00800000);
```

- Example where $n = 2$: we have $r = 4x$ and in this case

$$R = X + \min(|X|, 2^{p-1}) + \min(|X| \cdot 2, 2^{p-1}).$$

This now takes 4 cycles on the ST231:

```
0 absX = X & 0x7FFFFFFF;          X1 = X << 1;
1 min1 = min(absX, 0x00400000);   min2 = min(X1, 0x00400000);
2
3 Rgen = X + min1 + min2;
```

- Example where $n = 3$: we have $r = 8x$ and then

$$R = X + \min(|X|, 2^{p-1}) + \min(|X| \cdot 2, 2^{p-1}) + \min(|X| \cdot 2, 2^{p-2}) \cdot 2.$$

This takes 5 cycles on the ST231. Here the addition with ($\texttt{min3} \ll \texttt{1}$) is executed in one cycle thanks to the `sh1add` instruction.

```
absX = X & 0x7FFFFFFF;          X1   = X << 1;
min1 = min(absX, 0x00400000);   min2 = min(X1, 0x00400000);
                                min3 = min(X1, 0x00200000);

Rgen = X + min1 + min2 + (min3 << 1);
```

Detecting and handling large and NaN input. Since zero input is covered in Theorem 5.2, *special input* now means large or NaN input. More precisely, we now take for C_{spec} the following equation:

$$C_{\text{spec}} = C_{\text{large}} \vee C_{\text{nan}}, \tag{5.9}$$

where $C_{\text{large}} = [|x| \geqslant \Omega']$ and $C_{\text{nan}} = [x \text{ is NaN}]$, which are exactly the same as specified in §5.2.3. The property below gives the implementation of C_{spec}.

Property 5.3. $C_{spec} = [|X| \geqslant \iota(\Omega')]$.

Proof. First, let us prove that when C_{large} or C_{nan} holds, $|X| \geqslant \iota(\Omega')$ holds.

- When C_{large} is true, x is either a finite number with $|x| \geqslant \Omega'$ or $\pm\infty$. Thus, $|X| \geqslant \iota(\Omega')$.
- When C_{nan} is true, x is NaN. Then, we have $|X| > \iota(\infty) > \iota(\Omega')$.

On the other hand, assume that neither C_{large} nor C_{nan} holds. This implies x is a finite floating-point number and overflow does not occur, so that $|x| < \Omega'$ and $|X| < \iota(\Omega')$. □

The handling of the special input is easier than that in §5.2.3, as zero input is not handled here. The listing below shows how to detect and handle special input of **mul2** for the binary32 format and $\circ = \text{RN}$.

Listing 5.3: Detecting and handling special input of **mul2** for the binary32 format, $\circ = \text{RN}$.

```
absX = X & 0x7fffffff;
Cspec =  absX >= 0x7F000000;
if(Cspec){
  sigX = X - absX;
  Cnan  = absX > 0x7F800000;
  if(Cnan)    return  X | 0x00400000;
  else        return  sigX | 0x7F800000;
```

```
 7 }else{
 8   // handling generic input x of mul2
 9   return Rgen;
10 }
```

For other rounding modes, similarly to §5.2.3 only minor modifications of the code are needed: here it suffices to update only line 6 (the line which handles large input) according to the rounding mode.

- Rounding to zero ($\circ = \mathrm{RZ}$):

  ```
  Cinf = absX == 0x7F800000;
  if(Cinf) return sigX|0x7F800000; return sigX|0x7F7FFFFF;
  ```

  Notice that in this case we need to distinguish between finite and infinite input, which is done by the computation of `Cinf`.

- Rounding down ($\circ = \mathrm{RD}$):

  ```
  Cinf = X >= 0x7F800000;
  if(Cinf) return sigX|7F800000; return 0x7F7FFFFF;
  ```

  Here, `Cinf` holds for all the negative large floating-point finite numbers and $+\infty$. Thus, we return the encoding of $\pm\infty$ according to the sign of the input when `Cinf` holds, or otherwise that of Ω for all large finite positive numbers.

- Rounding up ($\circ = \mathrm{RU}$):

  ```
  Covf = (X + min(absX, 0x00800000)) >= 0xFF000000;
  if(Covf) return 0xFF7FFFFF; else return sigX|0x7F800000;
  ```

  Here `Covf` holds for large negative input. Then, we return the encoding of $-\Omega$ when `Covf` holds, or otherwise that of $\pm\infty$. Notice that the computation of `Covf` involves the result of the generic input and, in such a case, we let the compiler optimize the scheduling of the C codes.

Performances on the ST231. Table 5.3 and Table 5.4 summarize the performances of scaling by some small positive values of n for the binary32 and the binary64 formats on the ST231. Similarly to Table 5.2, we give both the latencies in cycles and the number of instructions (in brackets) for each rounding mode.

Comparing to Table 5.2, we see that up to $n = 4$ the specialized operators are faster than (or as fast as, for $\circ = \mathrm{RU}$) the general scaling operator with $n \geqslant 0$ for the binary32 format. However, for the binary64 format, the specialized operators do not give any performance gain as soon as $n \geqslant 3$.

The first line of Table 5.3 gives the performances of mul2 for the binary32 format, which is 3x to 4x faster than the general multiplication of FLIP 1.0 depending on the rounding modes.

n	RN		RU		RD		RZ	
1	5	[11]	7	[13]	6	[14]	6	[15]
2	7	[14]	8	[16]	7	[17]	7	[18]
3	7	[16]	9	[18]	8	[19]	8	[20]
4	8	[18]	10	[20]	8	[21]	9	[22]
5	9	[20]	10	[22]	10	[23]	9	[24]
6	9	[23]	12	[25]	11	[26]	10	[27]
7	10	[26]	13	[27]	11	[29]	11	[30]
8	11	[29]	14	[30]	12	[32]	12	[33]

Table 5.3: Latencies in # cycles [code sizes in # instructions] of scaling by some small values of n for the binary32 format.

n	RN		RU		RD		RZ	
1	10	[28]	12	[34]	11	[34]	11	[34]
2	12	[39]	14	[45]	14	[45]	14	[45]
3	15	[50]	17	[56]	16	[56]	16	[56]
4	18	[61]	20	[67]	20	[67]	19	[67]
5	21	[72]	23	[78]	23	[78]	23	[78]
6	24	[84]	27	[90]	26	[90]	26	[90]
7	28	[96]	30	[102]	29	[102]	29	[102]
8	31	[108]	33	[114]	33	[114]	32	[114]

Table 5.4: Latencies in # cycles [code sizes in # instructions] of scaling by some small values of n for the binary64 format.

Meanwhile, from Table 5.3, we can deduce that the IPCs of mul2 are around 2, which indicates that on average only two instructions are executed in each cycle. However, since several 32-bit constants are used in the C codes, the real bundle occupancies are very high. For instance, for mul2 of the binary32 format, the first four bundles are full when $\circ = \mathrm{RN}$.

5.3 Scaling by negative powers of two

In this section, we now assume

$$n < 0.$$

In this case, for every finite x, we have $|x| \cdot 2^n \leqslant |x|$, and therefore overflow will not occur. However, since the significand of the result is limited to p bits, the result can be inexact and rounding becomes necessary.

5.3.1 Specification

Since overflow cannot occur when $n < 0$, we do not need a refined specification as for $n \geqslant 0$ (see Table 5.1), and it is enough to consider (5.1). Another consequence of the absence of overflow is that the distinction between special and generic input is somehow simpler, and we have the following definition.

Definition 5.2. *For scaling and when $n < 0$, input x is called* **special** *when x is $\pm\infty$ or NaN, otherwise it is* **generic**.

In terms of condition C_{spec}, this means

$$C_{\text{spec}} = \big[x \text{ is } \pm\infty \text{ or NaN}\big]. \tag{5.10}$$

5.3.2 Scaling generic input

A normalized formula for $x \cdot 2^n$. Since x is generic, it is finite and therefore the exact result $\rho = x \cdot 2^n$ has the form

$$\rho = (-1)^{s_x} \cdot m_x \cdot 2^{e_x+n}.$$

Let us first consider separately two cases, depending on the value of $e_x + n$:

- If $e_x + n \geqslant e_{\min}$ then r is a floating-point number, so that no rounding error occurs.

- If $e_x + n < e_{\min}$ then by defining the positive integer

  $$\delta = e_{\min} - (e_x + n), \tag{5.11}$$

  we can *normalize* the exact result ρ as follows:

  $$\rho = (-1)^{s_x} \cdot m_x \cdot 2^{-\delta} \cdot 2^{e_{\min}},$$

  where $m_x \cdot 2^{-\delta}$ is in $[0, 1)$. In this case, $m_x \cdot 2^{-\delta}$ can have more than $p-1$ bits, so that rounding may be necessary.

Thanks to the definition of δ, these two cases can in fact be handled simultaneously by the following formula:

$$\rho = (-1)^{s_x} \cdot \ell \cdot 2^d, \tag{5.12a}$$

where

$$\ell = m_x \cdot 2^{-\max(\delta,0)}, \qquad d = e_{\min} - \min(\delta, 0). \tag{5.12b}$$

Note that by construction, we have either $\ell \in [0,1)$ and $d = e_{\min}$, or $\ell \in [1,2)$ and $d \in [e_{\min}, e_{\max}]$.

Implementation for the binary$k$ format. We detail here how to implement, for x generic and the binary$k$ floating-point format, the computation of scaling when $n < 0$ by using k-bit integer arithmetic and logic. Given the normalized result ρ as in (5.12), the standard encoding R of $\circ(\rho)$ can be deduced from Fact 2.1:

$$R = s_x \cdot 2^{k-1} + D \cdot 2^{p-1} + L + b,$$

where $D = d - e_{\min} =$, $L = \lfloor \ell \cdot 2^{p-1} \rfloor$, and b is specified by (2.10d).

$\boxed{\text{Computing } D \text{ and } L.}$ By (5.12b), we have $D = -\min(\delta, 0)$, and $L = \lfloor m_x \cdot 2^{-\max(\delta,0)} \cdot 2^{p-1} \rfloor$, where δ is a function of e_x. Instead of e_x, what we can extract directly from the encoding X is E_x. Then, we deduce the property below to compute D and L by using this integer E_x.

Property 5.4. *Let $\Delta = m_{x,0} - n$ and $M_x = m_x \cdot 2^{p-1}$. Then*

$$D = E_x - \min(\Delta, E_x).$$

and

$$L = \lfloor M_x \cdot 2^{-\mu} \rfloor,$$

where $\mu = \min(p+1, \max(\Delta, E_x) - E_x)$.

Proof. Since $n < 0$, we have $\Delta \in \{-n, 1-n\}$, from which we deduce $0 < \Delta \leqslant 2^{31} + 1$, which will be stored in an unsigned 32-bit integer. Furthermore, from $E_x = e_x - e_{\min} + m_{x,0}$, it follows that δ defined in (5.11) satisfies

$$\begin{aligned} \delta &= m_{x,0} - n - E_x \\ &= \Delta - E_x. \end{aligned}$$

Using (5.12b), we see that $D = d - e_{\min}$ is given by

$$\begin{aligned} D &= -\min(\delta, 0), \\ &= E_x - \min(\Delta, E_x), \end{aligned}$$

which is the desired expression for D.

Let us now prove the expression for L. Since $L = \lfloor \ell \cdot 2^{p-1} \rfloor$ and $\ell = m_x \cdot 2^{-\max(\delta,0)}$, we have

$$L = \lfloor M_x \cdot 2^{-(\max(\Delta, E_x) - E_x)} \rfloor,$$

and since $M_x < 2^p$, $L = \lfloor M_x \cdot 2^{-\mu} \rfloor$. □

Given the minu and maxu instructions, Property 5.4 can be implemented efficiently. For the binary32 format, shown in Listing 5.4, $m_{x,0}$ is computed in line 1 by variable m0, and Δ (variable Delta) is obtained in the next cycle. Then, both instructions minu and maxu are applied to Delta and Ex (computed in parallel with m0 in line 1). Hence, D is obtained in 5 cycles and L is obtained in 7 cycles.

$\boxed{\text{Computing } b.}$ Let us start by considering rounding to nearest even, for which $b = g \wedge (\ell_{p-1} \vee t)$ with g and t as defined in (2.10d). Since by definition of μ, we have $0 \leqslant \mu \leqslant p+1$, the computation of g can be implemented as shown in the property below:

Property 5.5. $g = G \bmod 2$ *with* $G = (2M_x) \gg \mu$.

Proof. First, g being the least significant bit of the integer $G = \lfloor \ell \cdot 2^p \rfloor$, we have $g = G \bmod 2$. Furthermore, by proceeding as in the proof of Property 5.4, one can deduce that $G = \lfloor 2M_x / 2^\mu \rfloor$. □

Since $\ell_{p-1} = L \bmod 2$, it suffices to get a formula for t. This is given by the next property.

Property 5.6. $t = [T \neq 0]$ *with* $T = (4M_x) \ll (k-1-\mu)$.

Proof. By definition of t, we have $t = [\{\ell \cdot 2^p\} \neq 0]$, and since m_x has at most p bits, one can replace ℓ by $m_x \cdot 2^{-\mu}$ in this identity, so that

$$\begin{aligned} t &= [\{m_x \cdot 2^{p-\mu}\} \neq 0] \\ &= [\{4M_x / 2^{\mu+1}\} \neq 0]. \end{aligned}$$

As $4M_x$ fits into a k-bit unsigned integer and the value $k-(\mu+1)$ is less than k, the fraction of $4M_x/2^{\mu+1}$ is zero if and only if the k-bit unsigned integer $T = 4M_x \ll (k-1-\mu)$ is zero. □

As shown in Listing 5.4, as soon as μ (variable mu at line 5) is obtained, it takes 2 cycles to compute T (variable T at line 7) and in these two cycles we can simultaneously compute L, G, g, and $L \,\&\, 1$. Then, b is computed at line 10. As a result, on the ST231 it takes 11 cycles to compute R (variable Rgen) for generic input, assuming the binary32 format and $\circ = \mathrm{RN}$.

Listing 5.4: Implementation of R for the binary32 format and $\circ = \mathrm{RN}$.

```
 0 absX = X & 0x7fffffff;                       F = X << 9;
 1 m0 = absX >= 0x00800000;                     Ex = absX >> 23;
 2 Delta =  (uint32_t)(m0-n);
 3 mup = maxu(Delta, Ex); nv = minu(Delta, Ex); Mx = (F>>9) | (m0<<23));
 4                        D = Ex-nv;            sigX = X- absX;
 5 mu = minu(25, mup-Ex); Mx2 = Mx << 1;        Mx4 = Mx << 2;
 6                        L = Mx >> mu;         G = Mx2 >> mu;
 7 T = Mx4 << (31-mu);    g = G & 1;
 8
 9 b = g && (T | ( L & 1));
10 Rgen =  (sigX | (D << 23)) + L + b;
```

Now, let us detail the implementations when $\circ$ is not RN. When the rounding mode is RZ, $b = 0$ as given in (2.10d). Consequently, the instructions involving g, G, T, and b can be suppressed and the last line of Listing 5.4 can be replaced by

```
Rgen =  (sigX | (D << 23)) + L;
```

For either RU or RN, recall that $g = \lfloor \ell \cdot 2^p \rfloor \bmod 2$ and $t = [\{\ell \cdot 2^p\} \neq 0]$, which are defined in (2.10d). Then, we have $g \vee t = [\{\ell \cdot 2^{p-1}\} \neq 0]$. By proceeding as in the proof of Property 5.6, we can deduce that $g \vee t = [T' \neq 0]$ with $T' = (2M_x) \ll (k - 1 - \mu)$. Therefore, we can remove the instructions for g, G, and T. For RU, line 9 of Listing 5.4 can be replaced by

```
b = (Mx2 << (31-mu)) && (sigX==0);
```

While for RD, this line is replaced by

```
b = (Mx2 << (31-mu)) && sigX;
```

5.3.3 Detecting and handling special input

By Definition 5.2, the handling of special input is to handle ±∞ or NaN input x. According to Equation (5.10), C_{spec} can be implemented as

$$C_{\text{spec}} = C_{\text{inf}} \vee C_{\text{nan}},$$

where $C_{\text{inf}} = [x \text{ is } \pm\infty]$, and $C_{\text{nan}} = [x \text{ is NaN}]$. In fact, as shown in (2.7), when x is not a finite floating-point number, the biased exponent field of the standard encoding X equals $2e_{\max} + 1$. Then, the property below gives the implementation of C_{spec}.

Property 5.7. $\mathrm{C}_{spec} = [E_x = 2e_{max} + 1]$, *where* E_x *is defined by* (2.4).

Proof. First, (2.7) gives that when C_{inf} or C_{nan} holds, $E_x = 2e_{\max} + 1$. Then, when neither C_{inf} nor C_{nan} holds, x is a finite floating-point number, and we have $E_x = e_x - e_{\min} + m_{x,0}$ with $e_x \in [e_{\min}, e_{\max}]$. Thus, $E_x \leqslant 2e_{\max}$. □

The listing below displays how to detect and handle special input of scaling when $n < 0$ for the binary32 format.

Listing 5.5: Detecting and handling special input of scaling when $n < 0$ for the binary32 format and $\circ = \text{RN}$.

```
absX = X & 0x7fffffff;
Ex = absX >> 23;
Cspec = Ex == 255;
if (Cspec){
   Cnan = absX > 0x7f800000;
   if(Cnan) return 0x7fc00000;
   else return X;
} else {
 8    // generic case (Listing 5.4)
 9    return Rgen;
10  }
```

In this case as well, since Listing 5.5 refers to Listing 5.4 at line 8, we have a complete code for scaling when $n < 0$, and we present the performances of this operator in the next paragraph.

Performances on the ST231. Table 5.5 gives the performances of scaling when $n < 0$ on the ST231. Since the latency for the binary32 format is 12 cycles when $\circ = \mathrm{RN}$ and we see from Listing 5.4 that it takes 11 cycles to handle the generic input, we can conclude that the implementation on the ST231 is optimally scheduled. For both formats, the IPCs are around 3, which indicates that on average two to three instructions are executed in each cycle.

In addition, the latencies of scaling by negative powers of two are longer than nonnegative scaling due to the expensive rounding scheme.

$\circ(x \cdot 2^n), n < 0$	RN	RU	RD	RZ
binary32	12 [33]	12 [30]	12 [29]	9 [23]
binary64	21 [66]	17 [56]	17 [56]	14 [41]

Table 5.5: Latencies in # cycles [code sizes in # instructions] for scaling when $n < 0$.

5.3.4 Specializing to some negative values of n

An algorithm based on case-selection. For generic x, we have seen at the beginning of §5.3.2 that $(\ell, d) = (m_x, e_x + n)$ if $e_x \geqslant e_{\min} + |n|$. In the other case, that is, when $e_x < e_{\min} + |n|$, we have also seen that $d = e_{\min}$ and ℓ is given by (5.12b); however, since e_x is lower bounded by $e_{\min}$, there are at most $|n|$ possible values for ℓ. More precisely, we have

$$(\ell, d) = \begin{cases} (m_x, e_x + n), & \text{if } e_x + n \geqslant e_{\min}, \\ (m_x \cdot 2^{-1}, e_{\min}), & \text{if } e_x + n = e_{\min} - 1, \\ (m_x \cdot 2^{-2}, e_{\min}), & \text{if } e_x + n = e_{\min} - 2, \\ \quad\vdots & \quad\quad\vdots \\ (m_x \cdot 2^{-|n|}, e_{\min}), & \text{if } e_x = e_{\min}. \end{cases} \tag{5.13}$$

Therefore, we can first compute the $|n| + 1$ floating-point numbers associated with the pairs (ℓ, d), and then use a binary tree of *select* instructions on the ST231 to return the correct result.

Example for $n = -1$. Here we give the implementation details for the **div2** operator, that is, how to get the encoding R of the result $\circ(x/2)$ from the encoding X of the input x.

Since x is generic, it is finite, and recalling Fact 2.1, we have

$$R = s_x \cdot 2^{k-1} + D \cdot 2^{p-1} + L + b$$

with $D = d - e_{\min}$ and $L = \lfloor \ell \cdot 2^{p-1} \rfloor$. We consider the two cases of (5.13) separately:

- If $e_x > e_{\min}$ then $(\ell, d) = (m_x, e_x - 1)$. In this case, the exact result $(-1)^{s_x} \cdot \ell \cdot 2^d$ is already a floating-point number. Hence, $b = 0$ and

  $$R = s_x \cdot 2^{k-1} + (e_x - e_{\min} + m_x) \cdot 2^{p-1} - 2^{p-1}.$$

  Thus, recalling (2.5), we conclude that $R = X - 2^{p-1}$.

- If $e_x = e_{\min}$ then $(\ell, d) = (m_x/2, e_{\min})$. In this case, $L = \lfloor M_x/2 \rfloor$ and $D = 0$. This leads to $R = s_x \cdot 2^{k-1} + \lfloor M_x/2 \rfloor + b$, and since $e_x = e_{\min}$ implies $|X| = M_x$, we arrive at

  $$R = s_x \cdot 2^{k-1} + (|X| \gg 1) + b. \tag{5.14}$$

  It remains to compute the round bit b. For doing this, note first that $\ell = m_x/2 = (\ell_0.\ell_1 \ldots \ell_p)_2$ has no more that p fraction bits, which implies that the sticky bit t is zero. Since the guard bit g is ℓ_p, the formula for b given in (2.10d) degenerates into

  $$b = \begin{cases} \ell_p \wedge \ell_{p-1}, & \text{if } \circ = \text{RN}, \\ \ell_p \wedge (\neg s_x), & \text{if } \circ = \text{RU}, \\ \ell_p \wedge s_x, & \text{if } \circ = \text{RD}, \\ 0, & \text{if } \circ = \text{RZ}. \end{cases}$$

  Now, we can easily check that

  $$\ell_p = X \bmod 2 \qquad \text{and} \qquad \ell_p \wedge \ell_{p-1} = [X \bmod 4 \neq 0].$$

  Consequently, the round bit b can be implemented by means of the following formula:

  $$b = \begin{cases} [X \bmod 4 \neq 0], & \text{if } \circ = \text{RN}, \\ (X \bmod 2) \wedge (\neg s_x), & \text{if } \circ = \text{RU}, \\ (X \bmod 2) \wedge s_x, & \text{if } \circ = \text{RD}, \\ 0, & \text{if } \circ = \text{RZ}. \end{cases} \tag{5.15}$$

To summarize, we have shown the following property:

Property 5.8. *If $n = -1$ and x is generic then*

$$R = \begin{cases} X - 2^{p-1}, & \textit{if } e_x > e_{min}, \\ \textit{as in (5.14) and (5.15)}, & \textit{otherwise.} \end{cases}$$

The corresponding C code for the binary32 format, and $\circ$ = RN is shown in Listing 5.6. In particular, note that detecting and handling special input are exactly the same as in § 5.3.3.

As shown at line 3 in Listing 5.6, `R1` corresponds to the result when $e_x > e_{\min}$, and `R2` corresponds to the result when $e_x = e_{\min}$, which dominates the cost of this operator. We see that `R1` takes only one cycle and can be computed in parallel with the computation of `R2`. As `R2` (obtained at line 3) has a latency of 4 cycles, the latency of R for generic input is of 5 cycles on the ST231.

Listing 5.6: Implementation of R for the binary32 format as by Property 5.8, $\circ = \mathrm{RN}$.

```
absX = X & 0x7FFFFFFF;
                       b = (X & 3) != 0;  Ex = absX >> 23; sigX = X - absX;
                       R1 = X - 0x00800000;                 Clarge = Ex > 1;
R2 = sigX + (absX >> 1) + b;
if (Clarge) R = R1; else R = R2;
```

Performances on the ST231. Table 5.6 and Table 5.7 summarize the performances of scaling by some specific negative values of n for the binary32 and the binary64 formats on the ST231.

By comparing with Table 5.5, we see that the specialized operators do not give any performance gain as soon as $n \leqslant -4$ for the binary32 format. Whereas for the binary64 format, the specialized operators are still slightly faster when $n = -4$ and $\circ = \{\mathrm{RN}, \mathrm{RZ}\}$. This is caused by the different emulations of 64-bit integer shifts used for the implementations of these two algorithms. The implementation of general scaling by negative powers of two for the binary64 format requires several general 64-bit integer shifts, which is costly on the ST231. However, the implementations for the specialized cases involve only 64-bit shifts by small constants, for which the compiler can generate fast emulations.

n	RN		RU		RD		RZ	
-1	6	[16]	8	[17]	7	[16]	6	[13]
-2	8	[24]	9	[24]	9	[23]	7	[17]
-3	11	[32]	11	[31]	11	[30]	9	[21]
-4	14	[40]	13	[38]	13	[38]	11	[25]

Table 5.6: Latencies in # cycles [code sizes in # instructions] of scaling by some negative values of n for the binary32 format.

n	RN		RU		RD		RZ	
-1	8	[26]	9	[27]	9	[27]	8	[21]
-2	12	[39]	12	[39]	12	[39]	9	[29]
-3	15	[52]	15	[51]	15	[51]	11	[37]
-4	18	[65]	18	[63]	18	[63]	13	[49]

Table 5.7: Latencies in # cycles [code sizes in # instructions] of scaling by some negative values of n for the binary64 format.

5.4 Complete implementation and experimental results

5.4.1 Scaling by an arbitrary power of two

So far, we have discussed different designs for scaling depending on the sign of n, since for nonnegative values of n, the result is exact but overflow may occur, whereas for negative values of n, there is no overflow but the result may be inexact. As in real applications we cannot always know the sign of n at compilation time, we discuss in this section the design of a full operator scaleB which handles both cases.

Complete implementation for an arbitrary power of two. Given the codes for scaling by nonnegative and negative powers of two described and analyzed in §§5.2 and 5.3, an implementation of a general scaling operator scaleB is straightforward: we essentially merge the two codes by introducing an *if-else* statement. Listing 5.7 displays such an implementation for the binary32 format.

Listing 5.7: ScaleB operator for the binary32 format and $\circ = \mathrm{RN}$.

```
if (n >= 0 && absX){
  //scaling when n is nonnegative and x is nonzero
  //code in Listing 5.2 with instructions involving Czero removed
}else{
  //scaling when n is negative or x is zero
  //code in Listing 5.5
}
```

Note that whatever the sign of n, zero input is handled by the C code used for the case $n < 0$ (Listing 5.5). This choice calls two remarks:

- First, let us see why this is correct. If we replace Δ in Property 5.4 by Δ', such that $\Delta' = (m_{x,0} - n) \bmod 2^{32}$, the new property holds for scaling by an arbitrary power of n for zero input as well as scaling when $n < 0$. For scaling when $n < 0$, since $0 < m_{x,0} - n \leqslant 2^{31} + 1$, we have $\Delta' = \Delta$, and therefore it falls to Property 5.4. When $n \geqslant 0$ and x is zero, we have $\Delta' \geqslant 0$ and $M_x = E_x = 0$. Therefore, we have $\mu \in [0, p+1]$ and $L = \lfloor M_x \cdot 2^{-\mu} \rfloor = 0$. Consequently, Properties 5.5 and 5.6 hold for an arbitary power of two when x is zero.

  Since we conclude in the proof of Property 5.4 that $0 < \Delta \leqslant 2^{31} + 1$ and that it will be stored in an unsigned 32-bit integer, we convert the result of `m0-n` explicitly to `uint32_t` at line 2 in Listing 5.4. Indeed, this conversion computes $(m_{x,0} - n) \bmod 2^{32}$, which is Δ'. Then, the code in Listing 5.5 can be applied to any n when x is zero.

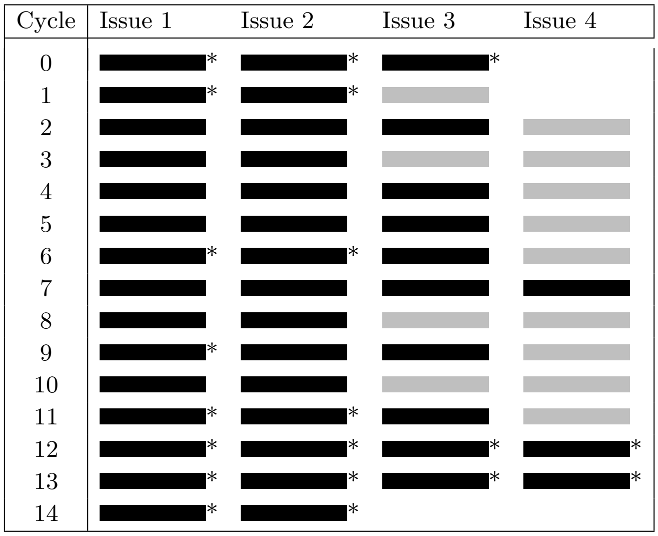
- Second, the advantage of such a choice is to save computational resources by removing the instructions of the special case handling zero input from scaling by nonnegative powers of two. For scaling when $n \geqslant 0$, a zero input is considered special to reduce the latency of the generic path (critical path) at the cost of more instructions; however, for scaling when $n < 0$, a zero input can be handled in the generic path without increasing the latency. Since both of the implementations in Listings 5.2 and 5.5 expose high ILP, to realize the ILP exposed by merging these two cases on the ST231, we need to reduce the total number of instructions for the lowest achievable latency.

Table 5.8 gives the performances of scaling by any powers of two. Ideally, the latency of scaleB for each rounding mode and binary$k$ format should be one cycle more than that of the corresponding case for scaling by negative powers of two, since scaling when $n < 0$ is more costly than scaling by nonnegative powers of two due to the rounding scheme. In practice, we see from Table 5.8 that although the overlaps are from three to five cycles for the binary32, the IPCs are over three in all these cases, which means that all the instructions are well scheduled.

		RN		RU		RD		RZ	
scaleB	binary32	15	[51]	16	[55]	16	[54]	14	[45]
	binary64	26	[96]	27	[98]	27	[98]	22	[77]

Table 5.8: Performances for scaleB: latencies in # cycles [code sizes in # instructions].

Figure 5.1 gives a precise description of the bundle occupancy when compiling the scaling implementation in Listing 5.7. The slots in black are those used to compute scaling when $n < 0$, those marked by $*$ are also used for scaling when $n \geqslant 0$, and those in grey are used for scaling when $n \geqslant 0$ only. Here, every constant longer than 9 bits occupies one slot beyond the slot used by the instruction operating on it. Among the 15 bundles, 13 bundles are fully used, which indicates a high ILP and thus a very good usage of the resources of the machine.



Cycle	Issue 1	Issue 2	Issue 3	Issue 4
0	■*	■*	■*	
1	■*	■*	▨	
2	■	■	■	▨
3	■	■	▨	▨
4	■	■	■	▨
5	■	■	■	▨
6	■*	■*	■	▨
7	■	■	■	■
8	■	■	▨	▨
9	■*	■	■	▨
10	■	■	▨	▨
11	■*	■*	■	▨
12	■*	■*	■*	■*
13	■*	■*	■*	■*
14	■*	■*		

Figure 5.1: Bundle occupancy for the scaling operator on ST231.

Regarding previous available software implementations, Cody and Coonen's algorithm [CC93] is fully based on other floating-point operators and it takes around 300-400 cycles depending on the inputs on the ST231. On the other hand, FDlibm supports only double precision inputs (binary64 format) and it is as fast as our implementation for binary64 normal numbers. As to subnormals, like for Cody and Coonen's algorithm, it depends on floating-point multiplication, which means that it will cost several hundreds of cycles in our context. In other words, the latencies for subnormal numbers of these algorithms are even larger than that of floating-point multiplication.

5.4.2 Application to matrix balancing

As already said in the introduction to this chapter, in linear algebra we often balance the input matrix to obtain more accurate eigenvalues. In particular, this balancing procedure is proceeded by default in MATLAB's function `eig` [HH05]. In radix 2, this amounts to replacing an $N \times N$ matrix A by DAD^{-1} with $D = \mathrm{diag}(d_1, \ldots, d_N)$ and each d_i is an integer power of two.

To evaluate the effect of scaling operators on matrix balancing, we use the code from [PTVF07, §11.6.1] shown in Listing 5.8. Here we observe that several multiplications can be specialized to mul2, mul4, div2 or div4, such as the multiplications at lines 13, 14, 18, and 19. Furthermore, we can fully rewrite the multiplications at lines 25 and 26 by the C11 standard scaling function `ldexp`. The complete re-written code is shown in Listing 5.9.

The test is run on the cycle-accurate simulator of the ST231. The baseline is by using the general floating-point operators of FLIP 1.0. The matrices are randomly generated by the C++ library function `rand` shown in the listing below.

```
time_t seconds;
int i;

time(&seconds);
srand((unsigned int) seconds);

cout<< "float A[N][N]={\\"  << endl;
for(i=0;i<N;i++){
   cout<< (float) rand()/((rand()%1000 + 1) << ",\\"  << endl;
}
cout << "};"<<endl;
```

With this code, when the generated floating-point number is not zero, it is at least 10^{-3}, which means these numbers are always in the normal range. However, since the performances of the scaling operators as well as the general operators of FLIP 1.0 do not depend on the input range, the constraint on the range of input will not affect the result of the experiment.

Figure 5.2 gives the speedups obtained for such $N \times N$ random matrices with N up to 100. The curve 'Speedup by specialized operators only' gives the speedups introduced by using the specialized operators mul2, mul4, div2, and div4 to compute the multiplications at lines 13, 14, 18, and 19 of Listing 5.8. The curve 'Speedup by fast scaling and specialized special operators' gives the speedups of balancing matrices by Listing 5.9, which uses both the scaleB operator and the specialized operators. We observe that the method using the scaleB operator leads to higher speedups, and since the accumulation of the loop (line 6 in both Listings 5.8 and 5.9) dominates the cost of balancing, both curves fall to the baseline as N approaches 100. However, for small dimensions, significant speedups are achieved thanks to our scaling operators. For example, in dimension $N = 10$, an acceleration of about 10% is obtained compared with FLIP 1.0.

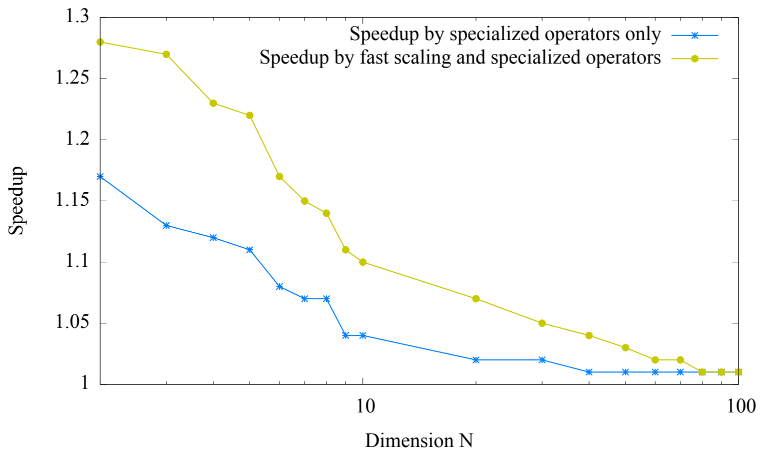


Figure 5.2: Speedups for balancing $N \times N$ matrices with $N \leqslant 100$.

Listing 5.8: Original C code for matrix balancing.

```
while(!done){
  done = true;
  for (i = 0; i < N; i++){
    float r = 0.0f, c = 0.0f;
    for(j = 0;j < N; j++)
      if(j != i){
        c += abs(A[j][i]);      r += abs(A[i][j]);
      }
    if(c != 0.0f && r != 0.0f){
      float g = r / 2.0f;
      float f = 1.0f;
      float s = c + r;
      while(c < g){
        f *= 2.0f;
        c *= 4.0f;
      }
      g = r * 2.0f;
      while (c > g){
        f /= 2.0f;
        c /= 4.0f;
      }
      if ((c + r) / f  < (0.95f) * s){
        done = false;
        g = (1.0f) / f;
        scale[i] *= f;
        for(j = 0; j < N; j++) A[i][j] *= g;
        for(j = 0; j < N; j++) A[j][i] *= f;
      }
    }
  }
}
```

Listing 5.9: Matrix balancing using `ldexp`.

```
while(!done){
  done = true;
  for (i = 0; i < N; i++){
    float r = 0.0f, c = 0.0f;
    for(j = 0; j < N; j++)
      if(j != i){
        c += abs(A[j][i]);      r += abs(A[i][j]);
      }
    if(c != 0.0f && r != 0.0f){
      float g = r / 2.0f;
      int exp = 0;
      float s = c + r;
      while(c < g){
        exp ++;
        c *= 4.0f;
      }
      g = r * 2.0f;
      while (c > g){
        exp --;
        c /= 4.0f;
      }
      if (ldexpf((c + r), 0-f) < (0.95f) * s){
        done = false;

        scale[i] = ldexpf(scale[i], exp);
        A[i][j] = ldexpf(A[i][j], -exp);
        A[j][i] = ldexpf(A[j][i], exp);
      }
    }
  }
}
```

Chapter 6

Two-dimensional dot products

Various real applications require the evaluation of floating-point two-dimensional dot products $xy + zt$. In this chapter, we study how to evaluate such expressions accurately and efficiently on VLIW integer processors. Accurately means that we provide correct rounding for the all the rounding modes as well as support for subnormal numbers; efficiently means that it shall be faster than evaluating the expressions by the naive approach consisting of two multiplications followed by one addition. For this, we propose an algorithm and its correctness analysis, which, like for the previous two chapters, is done in a parametrized way. We also detail the corresponding C implementation for the binary32 format. On the ST231, this code is from 1.15x to 1.3x faster than the naive approach. It also exposes a lot of ILP, with an IPC of at least 3.8. Furthermore, combining it with other custom operators leads to significant speedups: 1.59x when performing FFT and up to 1.45x for some 3D graphics applications.

6.1 Introduction

This chapter deals with the operator $(x, y, z, t) \mapsto xy + zt$, which evaluates the dot product of two vectors in dimension two:

$$xy + zt = \begin{bmatrix} x & z \end{bmatrix} \begin{bmatrix} y \\ t \end{bmatrix}.$$

Two-dimensional dot products (DP2) occur in several situations, and we list below a few examples:

- **Evaluation of discriminants.** When computing the roots of a quadratic equation $ax^2 + bx + c = 0$ for some given floating-point coefficients $a \neq 0$, b, c, one typically has to evaluate the discriminant

  $$b^2 - 4ac,$$

  which in radix 2 is of the form $xy + zt$ with x, y, z, t floating-point numbers such that, for instance, $x = y = b$, $z = -2a$, and $t = 2c$.

- **Complex arithmetic.** If $a = a' + ia''$ and $b = b' + ib''$ are two complex numbers whose real and imaginary parts are floating-point data, then multiplication and division naturally involve floating-point DP2s, possibly in a specialized form like sums of two squares:

  $$ab = a'b' - a''b'' + i\big(a''b' + a'b''\big),$$

and

$$\frac{a}{b} = \frac{a'b' + a''b''}{b'^2 + b''^2} + i\frac{a''b' - a'b''}{b'^2 + b''^2}.$$

Applications using complex arithmetic thus eventually rely on essentially floating-point addition, division, and DP2. This is for example the case of FFT computations, whose core block (the so-called 'butterfly' operation) involves DP2s and pairs of addition/subtraction [SS12].

Although the DP2 operation is not mentioned in the IEEE 754-2008 standard [IEE08], it has been studied both in hardware and in software. For example, hardware designs have been proposed in [SEES08] and [SS12], with applications to accelerating FFT computations. In software, it is known that DP2s can be implemented accurately using floating-point arithmetic, especially if a fused-multiply add (FMA) operation is available, using an algorithm due to Kahan [Kah98, Hig02, JLM12]. However, in our context, this approach would be too expensive and, on the other hand, Kahan's algorithm does not provide the correctly rounded result.

In this chapter, we study how to evaluate such expressions accurately and efficiently on VLIW integer processors. Accurately means that we provide correct rounding for the all the rounding modes as well as support for subnormal numbers; efficiently means that it shall be faster than evaluating the expressions by the naive approach consisting of two multiplications followed by one addition.

Our first contribution is to provide a specification for DP2 inspired by the IEEE specification of FMA, together with a definition of generic and special input that is well adapted to an efficient implementation.

Our second contribution is a detailed algorithm for DP2 that aims at high ILP exposure. As in the previous two chapters, this algorithm is described in a way parametrized by the input/output binary$k$ floating-point format. Although this algorithm reduces to a classical summation algorithm, it introduces a new swapping step, which is more economical in terms of *slct* instructions.

Third, as a consequence of this algorithm analysis, we obtain a complete C implementation for the binary32 format and each rounding mode. On the ST231, we get a latency of 55 cycles for rounding 'to nearest even'. This is 1.24x faster than 68-cycle latency of the naive implementation using two multiplications followed by one addition of FLIP 1.0; roughly similar performances are observed for the other rounding modes. As another consequence of our analysis, we also derive an implementation for fused multiply-add (FMA). In all cases, our codes expose a lot of ILP, with IPCs between 3.8 and 3.9 for DP2 and between 3.6 to 3.8 for FMA.

Finally, we analyze the practical impact of DP2 on two real applications, namely some FFTs and a 3D-graphics pipeline. Here we show that combining DP2 with other custom operators allows to accelerate these applications by factors of, respectively, 1.59 and up to 1.45.

Outline. First, we provide in §6.2 our specification for the DP2 operator, and deduce the definitions of generic and special input. Then, we detail in §§6.3 and 6.4 the algorithms and implementations for handling, respectively, generic and special input. Finally, §6.5 reports on the performances obtained on the ST231 for this DP2 operator alone, for its specialization to FMA, and for its application to the FFT and the 3D-graphics pipeline.

6.2 Specification

Since DP2 is not part of the standard, we first have to propose a specification for this operation. Roughly, our specification requires correct rounding for finite input, and for infinite or NaN input it follows the same rules as the ones for FMA in the standard. The two paragraphs below detail each of these aspects.

One rounding for the exact result ρ. For finite x, y, z, and t, DP2 shall compute $xy + zt$ as if with unbounded range and precision, and then round this exact value to the destination format. Thus, during this process, either only one rounding occurs or there is an overflow.

Overflow. To characterize it, let ρ denote the exact result $xy + zt$ rewritten in normalized form:

$$\rho = (-1)^s \cdot \ell \cdot 2^d,$$

where $s \in \{0, 1\}$ and where $(\ell, d) \in \mathbb{R} \times \mathbb{Z}$ satisfies

- either $\ell \in [0, 1)$ and $d = e_{\min}$,
- or $\ell \in [1, 2)$ and $d \geqslant e_{\min}$.

Using this rewriting, we see that overflow before rounding occurs if and only if

$$d > e_{\max}.$$

According to the specification of overflow in the standard [IEE08, §7.4], the result r to be returned satisfies

$$r = \begin{cases} (-1)^s \cdot \infty, & \text{if } \circ = \text{RN}\ , \\ (-1)^s \cdot \Omega, & \text{if } \circ = \text{RZ}, \\ \infty, & \text{if } \circ = \text{RU and } s = 0, \\ -\Omega, & \text{if } \circ = \text{RU and } s = 1, \\ \Omega, & \text{if } \circ = \text{RD and } s = 0, \\ -\infty, & \text{if } \circ = \text{RD and } s = 1. \end{cases} \tag{6.1}$$

Recall that $\Omega = (2 - 2^{1-p}) \cdot 2^{e_{\max}}$ is the largest normal floating-point number.

Sign bit when the exact result is zero. When overflow does not occur, we can apply Fact 2.1 to compute the standard encoding of the correctly-rounded result. When the exact result is zero ($\ell = 0$), we view $xy + zt$ as a sum of xy and zt, and apply the rules specified in the standard for floating-point addition [IEE08, §6.3]. This gives the two cases below:

- If xy and zt are either nonzero or zero with opposite signs, $s = 0$ for $\circ \in \{\text{RN}, \text{RU}, \text{RZ}\}$, and $s = 1$ for $\circ = \text{RD}$;
- If xy and zt are zero and of the same sign, s gets the sign of the products for $\circ \in \{\text{RN}, \text{RU}, \text{RZ}\}$, and $s = 1$ for $\circ = \text{RD}$.

Rules for infinite and NaN input. If any of input the x, y, z, or t is infinite or NaN, the result can only be either infinite or NaN.

$\boxed{\text{Returning NaN.}}$ The result is NaN if and only if one of the following statements is true:

- One of the input is NaN;
- One of the two products xy or zt is NaN;
- The sum of xy and zt is NaN;

The second statement holds when a multiplication is either $\infty \times 0$ or $0 \times \infty$. The third statement holds when an addition is the sum of two infinite numbers with opposite sign.

$\boxed{\text{Returning } \infty \text{ or } -\infty.}$ Meanwhile, when at least one of the input is infinity, and the result is not NaN, we return infinity as specified in the standard [IEE08, §6.1]:

- For multiplication, for finite or infinite $x \neq 0$, we have

$$(-1)^{s_{\text{inf}}}\infty \times (-1)^{s_x}|x| = (-1)^{s_{\text{inf}} \oplus s_x}\infty.$$

- For addition, for finite x, we have

$$(-1)^{s_{\text{inf}}}\infty + x = (-1)^{s_{\text{inf}}}\infty.$$

Definition for generic and special input. Thanks to the specification above, we give the definition below to classify generic and special input for DP2.

Definition 6.1. *For DP2, input is* **generic** *when for finite x, y, z, and t, the exact result is nonzero and overflow does not occur, and* **special** *otherwise.*

Note that an exact zero result is considered as special in order to simplify the handling of generic input. Like for all the other operators whose generic input is on the critical path, we try to move some computation to the special path in order to balance the costs of the two paths to decrease the whole latency.

Hence, using the definition above, the condition C_{spec} satisfies

$$C_{\text{spec}} = C_{\text{nan}} \vee C_{\text{inf}} \vee C_{\text{ovf}} \vee C_{\text{zero}}, \tag{6.2}$$

where

- $C_{\text{nan}} = [r \text{ is NaN}]$,
- $C_{\text{inf}} = [\text{one of } x,\ y,\ z,\ t \text{ is infinite}]$,
- $C_{\text{ovf}} = [\text{overflow occurs}]$,
- $C_{\text{zero}} = [\rho \text{ is } \pm 0]$.

6.3 Computing correctly-rounded 2D dot products for generic input

In this section we assume that the input (x, y, z, t) is generic, which means that the condition C_{spec} in §6.2 is false. Since in this case x, y, z, t must be finite floating-point numbers, they have the form $x = (-1)^{s_x} \cdot m_x \cdot 2^{e_x}$, $y = (-1)^{s_y} \cdot m_y \cdot 2^{e_y}$, $z = (-1)^{s_z} \cdot m_z \cdot 2^{e_z}$, $t = (-1)^{s_t} \cdot m_t \cdot 2^{e_t}$. Furthermore,

$$\text{either } xy \neq 0 \text{ or } zt \neq 0, \tag{6.3}$$

for otherwise we would have $xy + zt = 0$ and thus a returned result equal to zero, a case which is already covered by the special path; see the condition C_{zero} in §6.2.

Following what is sometimes done for other operators like addition, multiplication, or FMA [EL04, MBdD+10], we shall normalize those of these numbers which are subnormal, by removing the leading zeros of their significands. Consider for example input x. Then its significand m_x has the form

$$m_x = (\underbrace{0.00 \cdots 00}_{\lambda_x \text{ zeros}} 1 m_{x,\lambda_x+1} \cdots m_{x,p-1})_2, \tag{6.4}$$

and we shall rewrite x as

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

with

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

If x is normal then the pair (m_x, e_x) is left unchanged, while if it is subnormal then $m_x \in (0, 1)$ is transformed to $m'_x \in [1, 2)$ by decreasing $e_x = e_{\min}$ down to $e'_x < e_{\min}$. Finally, note that if x is zero then (6.4) gives $\lambda_x = p$, so that $m'_x = m_x = 0$ and $e'_x = e_{\min} - p$. In particular, we see that

$$e'_x \in [e_{\min} - p, e_{\max}].$$

Since we have a cheap (1 cycle) leading-zero count instruction available on the ST231, we can afford such normalization for the four components x, y, z, and t of the input.

Let us now define the following quantities, which are associated to the *exact* products xy and zt:

- their signs $s_{xy} = s_x \oplus s_y$ and $s_{zt} = s_z \oplus s_t$, with $\oplus$ denoting the logical XOR,
- the exact products of the normalized significands $m_{xy} = m'_x m'_y$ and $m_{zt} = m'_z m'_t$,
- the exponent sums $e_{xy} = e'_x + e'_y$ and $e_{zt} = e'_z + e'_t$.

Then, the exact result can clearly be rewritten as

$$xy + zt = (-1)^{s_{xy}} \cdot m_{xy} \cdot 2^{e_{xy}} + (-1)^{s_{zt}} \cdot m_{zt} \cdot 2^{e_{zt}}. \tag{6.5}$$

The identity in (6.5) simply says that DP2s, for generic input, can be seen as a kind of floating-point sum with precision $2p$ instead of p, and with exponent range

$[2e_{\min} - 2p, 2e_{\max}]$ instead of $[e_{\min}, e_{\max}]$. Thus, a natural approach to compute $\circ(xy + zt)$ will be to first normalize the input and prepare the above encodings for the two summands xy and zt, and then perform essentially a floating-point algorithm. This summation is itself usually composed of several steps [EL04, MBdD$^+$10]:

- Swap of the summands depending on their magnitude;
- Alignment of the operand of smallest magnitude to the one of largest magnitude;
- Fixed-point addition/subtraction;
- Normalization and rounding of the result.

The next subsections detail algorithms and implementations for all these steps.

6.3.1 Normalizing the input and preparing the two summands

This first step will consist in computing from the input encodings X, Y, Z, and T the following integer quantities:

- the bits s_{xy} and s_{zt};
- the $2k$-bit unsigned integers

  $$M_{xy} = N_x N_y \qquad \text{and} \qquad M_{zt} = N_z N_t,$$

  where for $i \in \{x, y, z, t\}$, N_i is the k-bit unsigned integer such that

  $$\begin{aligned} N_i &= m'_i \cdot 2^{k-2} \\ &= [0\,\underbrace{**\!*\cdots**\!*}_{p \text{ bits}}\underbrace{000\cdots000}_{w-1 \text{ zeros}}]; \end{aligned}$$

- the k-bit signed integers

  $$D_{xy} = e_{xy} - e_{\min} \qquad \text{and} \qquad D_{zt} = e_{zt} - e_{\min}. \tag{6.6}$$

Note that the triples (s_{xy}, M_{xy}, D_{xy}) and (s_{zt}, M_{zt}, D_{zt}) exactly encode the summands xy and zt, respectively. In particular, we have

$$\begin{aligned} M_{xy} &= m_{xy} \cdot 2^{2k-4} \\ &= [00\,\underbrace{****\cdots****}_{2p \text{ bits}}\underbrace{0000\cdots0000}_{2w-2 \text{ zeros}}], \end{aligned}$$

and similarly for M_{zt}.

Implementation of M_{xy}, D_{xy}, and s_{xy}. Now we detail the computation of (s_{xy}, M_{xy}, D_{xy}) and provide C codes for the binary32 format. In our implementation, the second triple (s_{zt}, M_{zt}, D_{zt}) is computed in exactly the same way.

$\boxed{\text{Computing } M_{xy}.}$ First, we should prepare the normalized significands m'_x and m'_y of x and y, by implementing the computation of N_x and N_y. Here we give only the details for N_x, since N_y is computed in the same way.

If $x \neq 0$, then

$$2N_x = \begin{cases} [1.m_{x,1}\dots m_{x,p-1}\overbrace{000\dots000}^{w \text{ zeros}}], & \text{if } x \text{ is normal},\\[2ex] [1.m_{x,\lambda_x+1}\dots m_{x,p-1}\underbrace{000\dots000}_{w+\lambda_x \text{ zeros}}], & \text{if } x \text{ is subnormal}. \end{cases}$$

We have seen in Chapter 2 that λ_x can be implemented as $\lambda_x = \max(\operatorname{clz}|X|, w) - w$, from which it follows that

$$2N_x = \big(X \ll \max(\operatorname{clz}|X|, w)\big) \mid 2^{k-1}.$$

Then, N_x is deduced as

$$N_x = \begin{cases} 2N_x \gg 1, & \text{if } |X| \neq 0,\\ 0, & \text{otherwise.} \end{cases}$$

Since the number of leading zeros can be computed efficiently on the ST231, the computation of $2N_x$ for the binary32 format (see variable `N2x` at line 4 in Listing 6.1) has a latency of 5 cycles. Since the ST231 is a 4-way VLIW processor, N_y can be computed in parallel and obtained at the same cycle as N_x.

Then, the ST231 instructions `mul32` and `mul64h` are used to compute the full 32×32 unsigned product (see Table 3.1). Since the two hardware multipliers can be used simultaneously, M_{xy} can be obtained in 3 cycles as soon as N_x and N_y are ready.

$\boxed{\text{Computing } D_{xy}.}$ Let $D_x = e_x - e_{\min}$ and $D_y = e_y - e_{\min}$. Since by definition $D_{xy} = e'_x + e'_y - e_{\min}$, we deduce that

$$\begin{aligned} D_{xy} &= e_x - \lambda_x + e_y - \lambda_y - e_{\min}\\ &= D_x + D_y - \lambda_x - \lambda_y + e_{\min}\\ &= D_x + D_y - \max(\operatorname{clz}|X|, w) - \max(\operatorname{clz}|Y|, w) + e_{\min} + 2w. \end{aligned}$$

Furthermore, D_x can be implemented as $D_x = E_x - m_{x,0}$ with $m_{x,0} = [\operatorname{clz}|X| \leqslant w]$, and the computation of D_y is exactly the same.

The computation of D_{xy} can be done fully in parallel with that of M_{xy} for the binary32 format on the ST231: see line 8 in Listing 6.1.

$\boxed{\text{Computing } s_{xy}.}$ In order to ease the packaging of the final result, what we compute in our implementation is $S_{xy} = s_{xy} \cdot 2^{k-1}$, which can easily be extracted from the input as follows:

$$S_{xy} = (X - |X|) \oplus (Y - |Y|).$$

This computation appears at lines 7 and 8 of Listing 6.1.

Listing 6.1: Implementation of M_{xy}, D_{xy}, and s_{xy} for the binary32 format.

```
0 absX = X & 0x7f800000;                 absY = Y & 0x7f800000;
1 lzx = clz(absX); lzy = clz(absY);      Sx = X - absX; Sy = Y - absY;
2 MX = max(lzx,8); MY = max(lzy,8);      mx0 = lzx <= 8;     my0 = lzy <= 8;
3
4 Nx2 = (X << MX) | 0x80000000; Ny2 = (Y << MY) | 0x80000000;
5                          Dx = (absX >> 23) - mx0; Dy = (absY >> 23) - my0;
6 if (absX != 0) Nx = Nx2 >> 1; else Nx = 0;
7 if (absY != 0) Ny = Ny2 >> 1; else Ny = 0;
8 Mxy = (uint64_t)Nx * (uint64_t)Ny; Dxy = Dx + Dy - MX - MY - 110; Sxy = Sx^Sy;
```

6.3.2 Swapping

Once the operands in the sum (6.5) are available, we may swap them in order to ensure that we will always add/subtract the one of smallest magnitude to the one of largest magnitude.

For this second step, let us define

$$\sigma = s_{xy} \oplus s_{zt}, \tag{6.7}$$

as well as the following quantities s_*, m_*, e_* for $* \in \{u, v\}$:

- $(s_u, m_u, m_v, e_v) = \begin{cases} (s_{xy}, m_{xy}, m_{zt}, e_{zt}) & \text{if } e_{xy} \geqslant e_{zt}, \\ (s_{zt}, m_{zt}, m_{xy}, e_{xy}) & \text{otherwise,} \end{cases}$

- $s_v = \sigma \oplus s_u$,

- $e_u = \begin{cases} \max(e_{xy}, e_{zt}) & \text{if } xyzt \neq 0, \\ e_{zt} & \text{if } xy = 0 \text{ and } zt \neq 0, \\ e_{xy} & \text{if } xy \neq 0 \text{ and } zt = 0. \end{cases}$

With this definition, we see that both m_u and m_v are in $[0, 4)$, with at least one of them being $\geqslant 1$. Furthermore, e_u and e_v are in $[2e_{\min} - 2p, 2e_{\max}]$, and in all cases

$$e_v = \min(e_{xy}, e_{zt}).$$

However, when one of the products is zero, e_u is not necessarily the maximum of e_{xy} and e_{zt}. For example, if $xy = 0$ then e_{xy} can be as large as $1 - p$ (since e'_x and e'_y are both at most $e_{\max}$, but one of them must be equal to $e_{\min} - p$) and e_{zt} can be as small as $2e_{\min} - 2p$, which is less that $1 - p$.

The next property shows that the exact result is preserved during such a swapping process. This is a priori not obvious due to the special definition of e_u.

Property 6.1. *For $* \in \{u, v\}$ let s_*, m_*, e_* be defined as above, and let*

$$u = (-1)^{s_u} \cdot m_u \cdot 2^{e_u} \quad \textit{and} \quad v = (-1)^{s_v} \cdot m_v \cdot 2^{e_v}.$$

Then the exact result satisfies

$$xy + zt = u + v.$$

Proof. The case where $xyzt \neq 0$ is clear. Assume now that $xy = 0$ (the case $zt = 0$ can be handled in exactly the same way, by exchanging indices). For this, we consider two subcases separately:

- If $e_{xy} \geqslant e_{zt}$ then (s_v, e_v, m_v) is equal to (s_{zt}, e_{zt}, m_{zt}) and $m_u = m_{xy}$ is zero. Thus, $(u, v) = (0, zt)$ and the result is true.
- If $e_{xy} > e_{zt}$ then, similarly, we can check that $(u, v) = (zt, 0)$, from which the result holds true as well.

□

Given the integers s_{xy}, s_{zt}, M_{xy}, M_{zt}, D_{xy}, D_{zt} as in §6.3.1, swapping will consist in producing

- the bits s_u and s_v;
- the $2k$-bit unsigned integers

$$M_u = m_u \cdot 2^{2k-4} \qquad \text{and} \qquad M_v = m_v \cdot 2^{2k-4};$$

- the k-bit signed integers

$$D_u = e_u - e_{\min} \qquad \text{and} \qquad D_v = e_v - e_{\min}.$$

Since m_u is one of m_{xy} and m_{zt}, the bit string of M_u has the same shape as the one of M_{xy} and M_{zt}:

$$M_u = [00\underbrace{****\cdots****}_{2p \text{ bits}}\underbrace{0000\cdots0000}_{2w-2 \text{ zeros}}],$$

and similarly for M_v.

Implementing the swap. We first need to evaluate the condition $e_{xy} \geqslant e_{zt}$, which by (6.6) is equivalent to

$$D_{xy} \geqslant D_{zt};$$

the corresponding C code for the binary32 format appears at line 0 of Listing 6.2.

$\boxed{\text{Computing } M_u,\ M_v \text{ and } s_u.}$ Since M_u and M_v are 64-bit integers, the expression below involves two `slct` instructions on the ST231.

```
if(Cns) Mu = Mxy else  Mu = Mzt;
```

Therefore, the computation of M_u and M_v at line 1 of Listing 6.2 costs four `slct` instructions.

By using S_{xy} and S_{zt}, we compute $S_u = s_u \cdot 2^{k-1}$ at line 1 as well, which costs one `slct` instruction on the ST231.

$\boxed{\text{Computing } D_u.}$ Since the input leading to an exact zero result is considered as special input, the computation of D_u falls to

$$D_u = \begin{cases} D_{zt} & \text{if } xy = 0, \\ D_{xy} & \text{if } zt = 0, \\ \max(D_{xy}, D_{zt}) & \text{otherwise.} \end{cases}$$

To evaluate whether the products xy and zt are zero, it suffices to evaluate the two conditions

$$[\min(|X|, |Y|) = 0] \qquad \text{and} \qquad [\min(|Z|, |T|) = 0];$$

the corresponding C code appears at lines 2 and 3 of Listing 6.2. Then, D_u is obtained at line 5 (see also lines 0 and 4).

The computations of D_v, σ, and $S_v = s_v \cdot 2^{k-1}$ are straightforward, whose corresponding C codes are at lines 0 and 2 of Listing 6.2.

Listing 6.2: Implementation of u and v for the binary32 format.

```
0 Du = max (Dxy, Dzt);   Dv = min(Dxy, Dzt);    Cns = Dxy >= Dzt; sigma = Sxy^Szt;
1 if (Cns) {Su = Sxy; Mu = Mxy; Mv = Mzt;} else {Su = Szt; Mu = Mzt; Mv = Mxy;}
2 M2 = min(absX, absY); M4 = min(absZ, absT); Sv = sigma ^ Su;
3 C2 = M2 == 0;          C4 = M4 == 0;
4 if (C2) Du = Dzt;
5 if (C4) Du = Dxy;
```

6.3.3 A normalized formula for $xy + zt$

The goal of this step is to express $xy+zt$ in the form $(-1)^s \cdot \ell \cdot 2^d$ with s, ℓ, d satisfying the conditions of applicability of Fact 2.1, that is, s is a bit and the pair (ℓ, d) is in $\mathbb{R} \times \mathbb{Z}$ and such that either $\ell \in [0, 1)$ and $d = e_{\min}$ (exact result in the subnormal range), or $\ell \in [1, 2)$ and $d \in [e_{\min}, e_{\max}]$ (exact result in the normal range). Note that overflow does not occur, since we are in the generic case.

To achieve this goal, we first align v with the exponent e_u, then we normalize this exponent to make it at least $e_{\min}$ and, finally, we normalize the resulting exact significand.

Alignment. Defining the integer

$$\delta = e_u - e_v, \tag{6.8}$$

and since σ in (6.7) satisfies $\sigma = s_u \oplus s_v$, we can rewrite the exact result as

$$xy + zt = (-1)^{s_u} \cdot \left(m_u + (-1)^{\sigma} \cdot m_v \cdot 2^{-\delta}\right) \cdot 2^{e_u}. \tag{6.9}$$

An important property of the integer δ is given below. It will be used in particular to prove the correctness of our implementation.

Property 6.2. *We have $\delta \geqslant 0$. Furthermore, if $m_u = 0$ then $\delta = 0$.*

Proof. The fact that δ is nonnegative comes from $e_u \in \{e_{xy}, e_{zt}\}$ and $e_v = \min(e_{xy}, e_{zt})$. Let us now check that it is zero when m_u is zero. In this case, either m_{xy} or m_{zt} must be zero but not both, because of (6.3). Assume for example that $m_{xy} = 0$ (the other case can be handled in exactly the same way by exchanging indices). Then, $xy = 0$ and thus

$$e_u = e_{zt}.$$

On the other hand, xy being zero implies zt is nonzero and thus $m_{zt} \neq 0$. Since $m_u = 0$ by assumption, this implies further that $m_u \neq m_{zt}$, so that we are in the case $e_{xy} \geqslant e_{zt}$. Recalling that e_v is the minimum of e_{xy} and e_{zt}, we deduce that

$$e_v = e_{zt}.$$

Hence $\delta = 0$. □

Exponent and significand normalization. Let us now define m as the exact sum in (6.9):

$$m = m_u + (-1)^{\sigma} \cdot m_v \cdot 2^{-\delta}. \tag{6.10}$$

First, it should be noticed that

$$m \neq 0,$$

for otherwise the returned result would be zero, which, as already said before, is not possible in the generic case (that is, when C_{spec} is false).

Then, denoting the sign and absolute value of m by s_m and $|m|$, respectively, we obtain

$$xy + zt = (-1)^{s_u \oplus s_m} \cdot |m| \cdot 2^{e_u}. \tag{6.11}$$

Since $|m|$ is nonnegative, we see that the sign s of both the exact result $xy + zt$ and of its rounded version $\circ(xy + zt)$ is given by

$$s = s_u \oplus s_m. \tag{6.12}$$

However, the expression in (6.11) is *not* normalized yet. Indeed, we have $m_u, m_v \in [0, 4)$, $\delta \geqslant 0$, and $m \neq 0$, so that all we can say for now is

$$|m| \in (0, 8), \tag{6.13a}$$

and

$$e_u \in [2e_{\min} - 2p, 2e_{\max}]. \tag{6.13b}$$

The two ranges in (6.13) are too wide, since we are looking for ℓ and d such that $\ell \in [0, 2)$ and $d \in [e_{\min}, e_{\max}]$. How to rescale $|m|$ and e_u in (6.11) in order to get a suitable pair (ℓ, d) is given by the next theorem.

Theorem 6.1. *Assume the input (x, y, z, t) is generic, and let the integers μ, λ, ν be defined as follows:*

- $\mu = \max(e_u, e_{min})$,

- *λ is the unique integer such that $|m| \cdot 2^{e_u - \mu} \in [2^{-\lambda}, 2^{-\lambda+1})$,*
- *$\nu = \min(\lambda, \mu - e_{min})$.*

Furthermore, let

$$\ell = |m| \cdot 2^{e_u - \mu + \nu} \qquad \textit{and} \qquad d = \mu - \nu.$$

Then

$$xy + zt = (-1)^s \cdot \ell \cdot 2^d$$

with s as in (6.12) and with (ℓ, d) such that

- *either $\ell \in [0, 1)$ and $d = e_{min}$,*
- *or $\ell \in [1, 2)$ and $d \in [e_{min}, e_{max}]$.*

Proof. By definition, $d = \mu - \min(\lambda, \mu - e_{\min})$, so that $d \geqslant \mu - (\mu - e_{\min}) = e_{\min}$. Let us now consider two cases:

- If $d = e_{\min}$ then it suffices to check that $\ell \in [0, 2)$. For this, note first that $\nu = \mu - e_{\min}$ by definition of d, which by definition of ν implies $\lambda \geqslant \mu - e_{\min}$. Now, using the definition of λ, we deduce that

  $$|m| \cdot 2^{e_u - \mu} < 2^{e_{\min} - \mu + 1},$$

  and then $\ell < 2^{e_{\min} - \mu + \nu + 1} = 2$, as wanted.
- If $d > e_{\min}$, let us show that ℓ is in $[1, 2)$. We have $\nu = \mu - d \leqslant \mu - e_{\min} - 1$ and thus $\nu = \lambda$. Then, by using the definition of λ, we deduce that $\ell = |m| \cdot 2^{e_u - \mu + \nu}$ is in $[2^{\nu - \lambda}, 2^{\nu - \lambda + 1}) = [1, 2)$.

□

Note first that the definition of μ implies

$$\mu - e_u \geqslant 0. \tag{6.14}$$

In Theorem 6.1 this nonnegative integer is used to normalize the exponent e_u in the case where it is below $e_{\min}$. Then, the other key integer is ν, which can be either positive or nonnegative, and whose effect will be to rescale $|m| \cdot 2^{e_u - \mu}$. However, as the next property shows, as soon as μ differs from e_u then this scaling factor ν is known to be at most zero.

Property 6.3. *If $\mu - e_u \neq 0$ then $\nu \leqslant 0$.*

Proof. The definition of μ implies that $\mu = e_{\min}$, from which it follows that $\nu = \min(\lambda, 0) \leqslant 0$. □

We conclude this section with another property, which says that if e_u is small enough then ν is zero and the normalized real mantissa is less than 2^{-p}.

Property 6.4. *If $\mu - e_u \geqslant p + 3$ then $\nu = 0$ and $\ell < 2^{-p}$.*

Proof. Since $\mu \neq e_u$, we have $\mu = e_{\min}$ and the definition of ν then implies

$$\nu = \min(\lambda, 0).$$

Now, by definition of λ and since $|m| < 8$, we have $2^{-\lambda} \leqslant |m| \cdot 2^{e_u - \mu} < 8 \cdot 2^{-p-3} = 2^{-p}$, so that

$$\lambda \geqslant 0.$$

Consequently, $\nu = 0$ and $\ell = |m| \cdot 2^{e_u - \mu} < 2^{-p}$. □

6.3.4 Implementation of $\circ(xy + zt)$ for the binary$k$ format

Theorem 6.1 makes it possible to apply Fact 2.1 and thus to recover the encoding R of $\circ(xy + zt)$ as

$$R = s \cdot 2^{k-1} + D \cdot 2^{p-1} + L + b,$$

where the sign bit s is as in (6.12) and where D, L, and b are as follows: $D = d - e_{\min}$, $L = \lfloor \ell \cdot 2^p \rfloor$, and the round bit b is defined as shown in (2.10d) in terms of ℓ_{p-1} (least significant bit of L), g (guard bit, equal to ℓ_p), and the sticky bit $[\{\ell \cdot 2^p\} \neq 0]$.

In the paragraphs below we detail how to get s, D, L, b from s_u, σ, M_u, M_v, D_u, D_v. As can be expected, the computations of L and of the sticky bit used for b are the most costly ones.

We proceed as follows: we shall first compute an approximation to the absolute value of m in (6.10). Then, this approximation will be useful for getting both the sticky bit and the integer G given by

$$G = \lfloor \ell \cdot 2^p \rfloor.$$

Finally, it will be easy to deduce

$$L = \lfloor G/2 \rfloor, \quad \ell_{p-1} = L \bmod 2, \quad g = G \bmod 2.$$

simultaneously from G.

Computing a $2k$-bit approximation M to the exact sum $|m|$. From (6.10) we have

$$|m| = \begin{cases} m_u + m_v \cdot 2^{-\delta} & \text{if } \sigma = 0, \\ m_u - m_v \cdot 2^{-\delta} & \text{if } \sigma = 1 \text{ and } m_u \geqslant m_v \cdot 2^{-\delta}, \\ m_v \cdot 2^{-\delta} - m_u & \text{if } \sigma = 1 \text{ and } m_u < m_v \cdot 2^{-\delta}, \end{cases} \tag{6.15}$$

where $m_u = M_u / 2^{2k-4}$, $m_v = M_v / 2^{2k-4}$, and $\delta = e_u - e_v = D_u - D_v$.

Our goal here is to define a suitable $2k$-bit integer M such that $M/2^{2k-4}$ approximates $|m|$ well enough, in a sense that will be made precise in Theorem 6.2.

To do this, we first need an equivalent integer formulation of the inequality $m_u < m_v \cdot 2^{-\delta}$ introduced in (6.15) to define $|m|$. Such a formulation is given by the next property.

Property 6.5. *Let*

$$\delta' = \min(\delta, 2p+2). \tag{6.16}$$

Then

$$m_u < m_v \cdot 2^{-\delta} \quad \textit{if and only if} \quad M_u < \lfloor M_v \cdot 2^{-\delta'} \rfloor.$$

Proof. Assume that $m_u = 0$. Then Property 6.2 implies $\delta = 0$ and (6.16) leads to $\delta' = 0$. Hence $\lfloor M_v \cdot 2^{-\delta'} \rfloor$ is equal to $M_v \cdot 2^{-\delta}$, and the conclusion follows from the fact that $M_u = m_u \cdot 2^{2k-4}$ and $M_v = m_v \cdot 2^{2k-4}$.

Assume now that $m_u > 0$ and consider the two implications separately:

- If $m_u < m_v \cdot 2^{-\delta}$ then, since $m_u \geqslant 1$ and $m_v < 4$, we deduce that $1 < 2^{2-\delta}$, that is, $\delta \leqslant 1$. Hence $\delta' = \delta \leqslant 1$ and thus $m_u < m_v \cdot 2^{-\delta}$ implies $M_u < M_v \cdot 2^{-\delta'} = \lfloor M_v \cdot 2^{-\delta'} \rfloor$, as wanted.
- Conversely, if $M_u < \lfloor M_v \cdot 2^{-\delta'} \rfloor$ then $M_u < M_v \cdot 2^{-\delta'}$. By dividing both sides by 2^{2k-4}, we deduce that $1 \leqslant m_u < m_v \cdot 2^{-\delta'} < 4 \cdot 2^{-\delta'}$. Hence $\delta' \leqslant 1$. Consequently, $M_v \cdot 2^{-\delta'}$ is an integer and $M_u < M_v \cdot 2^{-\delta'}$ can be rewritten as $M_u < \lfloor M_v \cdot 2^{-\delta'} \rfloor$.

□

Then we need also an integer version of the sums defining the possible values of $|m|$ in (6.15). Here, approximation will be necessary, since $|m|$ may not be representable using $2k$ consecutive bits. Taking

$$T_{\delta'} = [M_v \bmod 2^{\delta'} \neq 0] \qquad \text{and} \qquad M'_v = \lfloor M_v \cdot 2^{-\delta'} \rfloor \,\big|\, T_{\delta'},$$

we approximate $|m|$ by $M/2^{2k-4}$, where M is the following $2k$-bit unsigned integer:

$$M = \begin{cases} M_u + M'_v & \text{if } \sigma = 0, \\ M_u - M'_v & \text{if } \sigma = 1 \text{ and } M_u \geqslant \lfloor M_v \cdot 2^{-\delta'} \rfloor, \\ M'_v - M_u & \text{if } \sigma = 1 \text{ and } M_u < \lfloor M_v \cdot 2^{-\delta'} \rfloor. \end{cases} \tag{6.17}$$

Since M_u and M_v are in $[0, 2^{2k-2}]$, one can check easily that M is in $[0, 2^{2k-1}]$. Consequently, we can write

$$\begin{aligned} M &= [0 * * * \overbrace{* * * * \cdots * * * *}^{2k-4 \text{ bits}}] \\ &\approx |m| \cdot 2^{2k-4}. \end{aligned}$$

The reason for which here we have replaced δ by δ' is the same as for implementing floating-point addition: indeed, as explained in [MBdD+10, p. 337], the shift value δ can be huge (in the hundreds for binary32) and, on the other hand, correct rounding does not require to shift by more than, roughly, the current precision (p for binary$k$ addition, $2p$ in the case of binary$k$ DP2) as soon as we can tell, for a huge value of δ, whether the quantity to be shifted is zero or not.

A case where δ is known to be small (and thus to be equal to δ') is indicated in the property below. This result will turn out to be useful for deriving our implementation of the computation of $G = \lfloor \ell \cdot 2^p \rfloor$ in the next paragraph.

Lemma 6.1. *If $M_u < M'_v$ then $\delta' = \delta \leqslant 1$.*

Proof. If $M_u = 0$ then $m_u = 0$, and Property 6.2 implies $\delta = 0$, and thus $\delta' = \min(0, 2p+2) = 0$.

Assume now $M_u > 0$. In this case, due to the normalization of input, we must have $M_u \geqslant 2^{2k-4}$. On the other hand, by definition, $M'_v \leqslant M_v \cdot 2^{-\delta'} + 1$ with $M_v = m_v \cdot 2^{2k-4} < 2^{2k-2}$. Therefore, when $M_u < M'_v$ we have

$$2^{2k-4} \leqslant M_u \leqslant M'_v - 1 < 2^{2k-2-\delta'},$$

which implies $\delta' \leqslant 1$. □

$\boxed{\text{Computing } \delta'.}$ First, let us discuss the computation of δ'. By using the k-bit signed integers $D_u = e_u - e_{\min}$ and $D_v = e_v - e_{\min}$, which are computed in Listing 6.2, and the definition δ, $\delta = e_u - e_v$, we have

$$\delta = D_u - D_v.$$

Therefore, we can obtain $\delta' = \min(\delta, 2p+2)$, which is implemented at line 1 of Listing 6.4.

$\boxed{\text{Implementing } M'_v.}$ The computation of M'_v requires $\lfloor M_v \cdot 2^{-\delta'} \rfloor$ and $T_{\delta'}$. Since $0 \leqslant \delta' \leqslant 2p+2 < 2k$ and M_v is a $2k$-bit unsigned integer, $\lfloor M_v \cdot 2^{-\delta'} \rfloor$ can be directly implemented by a right shift as follows:

$$M_v \gg \delta',$$

which appears at line 7 of Listing 6.4.

Now, let us discuss the computation of $T_{\delta'}$, which is more involved. A direct way to implement $T_{\delta'}$ could be

$$T_{\delta'} = [(M_v \ll (2k - \delta')) \neq 0].$$

However, since $2k - \delta' \in [2w - 2, 2k]$ and the behavior of the shift operators is undefined in the C standard if the value of the right operand equals the width of the left operand, the implementation above is not safe.

Then, there are two alternative solutions:

- We can use an *if-else* statement to implement $T_{\delta'}$ as follows:

$$T_{\delta'} = \begin{cases} [(M_v \ll (2k - \delta')) \neq 0], & \text{if } \delta' \neq 0, \\ 0, & \text{otherwise.} \end{cases}$$

  For the implementation of the binary32 format on the ST231, since the emulation of the 64-bit shift has a latency of 4 cycles (see Table 3.3 and, for a detailed algorithm, see §3.4.1), this method costs 6 cycles: that is, 4 cycles for the shift, 1 cycle for the equality operator (`!=`) for $[(M_v \ll (2k - \delta')) \neq 0]$, and 1 cycle for the select instruction (`slct`). The condition $[\delta' \neq 0]$ can be computed in parallel and is not counted for the final latency.

- Since $\delta' \in [0, 2p+2]$, we can also split the shift in two parts as follows:

  $$T_{\delta'} = [((M_v \ll 1) \ll (2k - 1 - \delta')) \neq 0].$$

  For the binary32 format on the ST231, realizing the second method by two 64-bit shifts can be costly. However, by using further the fact that the last significant bit of the integer M_v is zero, we can efficiently incorporate the two shifts into the equality operator and obtain $T_{\delta'}$ in 5 cycles. The corresponding C code is given in Listing 6.3.

Listing 6.3: Implementation of $T_{\delta'}$ for the binary32 format.

```
uint32_t compute_Td(uint64_t Mv, uint32_t deltap){
  uint32_t TdH,TdL,Td,Lv,Hv;
  int32_t A,A1,A2,B,B1;

  Lv = (uint32_t) Mv;         Hv = Mv >> 32;

  A = 32 - deltap;            B = 63 - deltap;
  A1 = min(A,31);             B1 = min(B,31);
  A2 = max(0,A1);

  TdL =  Lv << A2;            TdH = (Hv << 1) << B1;

  Td = TdL || TdH;
  return Td;}
```

When both $\lfloor M_v \cdot 2^{-\delta'} \rfloor$ and $T_{\delta'}$ are ready, M'_v is realized by a bitwise OR of these two parts, shown at line 7 of Listing 6.4.

Implementing the condition $[M_u < \lfloor M_v \cdot 2^{-\delta'} \rfloor]$. When $\lfloor M_v \cdot 2^{-\delta'} \rfloor$ is available, the implementation this condition is straightforward. Here we introduce another way to implement this condition based on the following property.

Property 6.6. *The condition* $[M_u < \lfloor M_v \cdot 2^{-\delta'} \rfloor]$ *is equivalent to* $[M_u < M'_v]$.

Proof. If $M_u < \lfloor M_v \cdot 2^{-\delta'} \rfloor$, by Property 6.5, we can deduce $\delta' \leqslant 1$. Then, we have $T_{\delta'} = 0$ and $M'_v = \lfloor M_v \cdot 2^{-\delta'} \rfloor$. Conversely, if $M_u < M'_v$, by Lemma 6.1, we have $\delta' \leqslant 1$. Again, we have $T_{\delta'} = 0$ and $M'_v = \lfloor M_v \cdot 2^{-\delta'} \rfloor$. □

The advantage of using the condition $[M_u < M'_v]$ (implemented at line 11 of Listing 6.4) is to decrease the liveness time of the register which holds the integer $\lfloor M_v \cdot 2^{-\delta'} \rfloor$. This can ease the instruction scheduling and leads to better ILP. In practice, for the binary32 format and on the ST231, one cycle is saved for the full DP2 operator.

Listing 6.4: Implementation of M for the binary32 format.

```
delta = Du - Dv;
deltap = min(delta, 50);




Td = compute_Td (Mv, deltp); //Listing 6.3
Mvp = (Mv >> deltap) | Td;


              Ma = Mu + Mvp;
if (Mu < Mvp) Ms = Mvp - Mu; else Ms = Mu - Mvp;
if (sigma) M = Ms; else M = Ma;
```

Computing ν and G. Recall from Theorem 6.1 that the integer ν is defined as

$$\nu = \min(\lambda, \mu - e_{\min}).$$

Here, λ depends on the exact sum $|m|$ and is thus not directly available, since we have only the approximation M. The theorem below gives expressions for both ν and G, in terms of this integer M. It uses in particular the clz function introduced in Chapter 2, which counts leading zeros: if X is a k-bit unsigned integer then

$$\operatorname{clz}(X) = \begin{cases} k & \text{if } X = 0, \\ k - \lfloor \log_2 X \rfloor - 1 & \text{otherwise.} \end{cases}$$

Theorem 6.2. *Let M be as in (6.17) and let the integers ϵ_1, $\widetilde{M}$, and λ' be defined as follows:*

- $\epsilon_1 = \min(\mu - e_u, p + 3)$,
- $\widetilde{M} = \lfloor M/2^{\epsilon_1} \rfloor$,
- $\lambda' = clz(\widetilde{M}) - 3$.

Then

$$\nu = \min(\lambda', \mu - e_{min}) \quad \textit{and} \quad G = \lfloor \widetilde{M} \cdot 2^{-\epsilon_2 - k} \rfloor,$$

where $\epsilon_2 = w - 4 - \nu$.

Proof. Assume first that $\mu - e_u \geqslant p + 3$. Then $\mu \neq e_u$, which implies $\mu = e_{\min}$ and then $\nu = \min(\lambda, 0)$. Now, by definition of λ and since $|m| < 8$, we have $2^{-\lambda} < 8 \cdot 2^{p-3} = 2^{-p}$. This implies $\lambda \geqslant 0$, and we conclude that $\nu = 0$. On the other hand, since $M < 2^{2k-1}$ and $\epsilon_1 = p + 3$, we have

$$\widetilde{M} \leqslant M \cdot 2^{\epsilon_1} < 2^{2k-1-p-3} = 2^{k+w-4}. \tag{6.18}$$

Consequently, $\mathrm{clz}\big(\widetilde{M}\big) \geqslant 2k - (k + w - 4) = p + 4$. Hence $\lambda' \geqslant p + 1$, which implies that $\min(\lambda', \mu - e_{\min})$ is zero and thus equal to ν.

Let us now consider G. From Property 6.4, it must be zero when $\mu - e_u \geqslant p + 3$. Since $\nu = 0$, we have $\epsilon_2 = w - 4$ and using (6.18) gives $\widetilde{M} \cdot 2^{-\epsilon_2 - k} < 1$. Hence its integer part is zero, and thus equal to G.

The rest of the proof deals with the case where

$$\mu - e_u < p + 3,$$

and for M in (6.17) we assume $\sigma = 1$ and $M_u < \lfloor M_v \cdot 2^{-\delta'} \rfloor$. (The other two cases can be handled using the same kind of reasoning.)

First, note that Lemma 6.1 implies $\delta' = \delta \leqslant 1$. Hence $M_v \cdot 2^{-\delta'}$ is an integer and it follows that $T_{\delta'} = 0$ and $M'_v = M_v \cdot 2^{-\delta}$. Therefore,

$$\begin{aligned} M &= M_v \cdot 2^{-\delta} - M_u \\ &= (m_v \cdot 2^{-\delta} - m_u) \cdot 2^{2k-4} \\ &= |m| \cdot 2^{2k-4}. \end{aligned}$$

Since $M \neq 0$, this implies

$$\begin{aligned} \mathrm{clz}(\widetilde{M}) &= \epsilon_1 + \mathrm{clz}(M) \\ &= \epsilon_1 + 2k - \lfloor \log_2 |m| + 2k - 4 \rfloor - 1 \\ &= 3 + \mu - e_u - \lfloor \log_2 |m| \rfloor \\ &= 3 + \lambda. \end{aligned}$$

Hence $\lambda' = \lambda$ and we conclude that ν equals $\min(\lambda', \mu - e_{\min})$.

Let us now consider $G = \lfloor \ell \cdot 2^p \rfloor$. Since $\epsilon_1 = \mu - e_u$, we have

$$\begin{aligned} G &= \lfloor |m| \cdot 2^{-\epsilon_1 + \nu + p} \rfloor \\ &= \lfloor M \cdot 2^{-\epsilon_1 + 4 - 2k + \nu + p} \rfloor \\ &= \lfloor M \cdot 2^{-\epsilon_1 - \epsilon_2 - k} \rfloor. \end{aligned}$$

We conclude by distinguishing between two cases:

- If $e_u > e_{\min}$ then $\epsilon_1 = 0$. Hence $\widetilde{M} = M$ and we arrive at the claimed formula for G.

- If $e_u \leqslant e_{\min}$ then $\mu = e_{\min}$, which implies $\nu \leqslant 0$. Hence both ϵ_1 and $\epsilon_2 + k$ are nonnegative, and then gives $G = \big\lfloor \lfloor M \cdot 2^{-\epsilon_1} \rfloor \cdot 2^{-\epsilon_2 - k} \big\rfloor$, as claimed.

□

Implementing $\widetilde{M}$. First, we should compute $\mu - e_u$ to get ϵ_1. Since $\mu - e_u = (\mu - e_{\min}) - (e_u - e_{\min})$ and the signed integer $D_u = e_u - e_{\min}$ has already been computed, we have

$$\mu - e_u = D' - D_u$$

with $D' = \mu - e_{\min}$. By definition, $\mu = \max(e_u, e_{\min})$, we deduce that

$$\begin{aligned} D' &= \max(e_u, e_{\min}) - e_{\min} \\ &= \max(D_u, 0). \end{aligned}$$

Hence, we have

$$\epsilon_1 = \min(D' - D_u, p + 3),$$

which is implemented at line 2 of Listing 6.5 for the binary32 format.

Then, the computation of $\widetilde{M}$ is straightforward:

$$\widetilde{M} = M \gg \epsilon_1.$$

For the binary32 format, this formula leads to a 64-bit shift, which takes 4 cycles on the ST231. However, since in this case ϵ_1 satisfies $0 \leqslant \epsilon_1 \leqslant 27 < 32$, this shift can be implemented as

$$\begin{aligned} \widetilde{M}_H &= M_H \gg \epsilon_1, \\ \widetilde{M}_L &= ((M_H \ll 1) \ll \widetilde{\epsilon_1}) \,|\, (M_L \gg \widetilde{\epsilon_1}), \end{aligned}$$

where $\widetilde{M} = \widetilde{M}_H \cdot 2^{32} + \widetilde{M}_L$, $M = M_H \cdot 2^{32} + M_L$, and

$$\widetilde{\epsilon_1} = 31 - \epsilon_1.$$

This is implemented at lines 3 to 5 of Listing 6.5.

$\boxed{\text{Implementing } G.}$ First, let us compute ν and ϵ_2. By defining `clzl` as a function that counts the number of leading zeros of the $2k$-bit integer $\widetilde{M}$, we have

$$\nu = \min(\operatorname{clzl} \widetilde{M} - 3, D').$$

This is implemented at line 10 of Listing 6.5. When $\widetilde{M}$ is a 64-bit integer, the function `clzl` is implemented by the intrinsic operator `__lzcntl` (see §3.1.5) on the ST231.

Then, the implementation of $\epsilon_2 = w - 4 - \nu$ is straightforward and we have

$$G = \begin{cases} \widetilde{M}_H \gg \epsilon_2 & \text{if } \epsilon_2 > 0, \\ \left(\widetilde{M} \ll -\epsilon_2\right) \gg k & \text{if } \epsilon_2 \leqslant 0. \end{cases}$$

Recall that $\widetilde{M}_H$ is the upper half of the $2k$-bit integer $\widetilde{M}$. The corresponding C code to compute ϵ_2 and G for the binary32 format is shown at lines 11 to 16 of Listing 6.5.

$\boxed{\text{Implementing } b.}$ The most difficult part for the implementation of b is how to compute the sticky bit τ, which is

$$\tau = [\{\ell \cdot 2^p\} \neq 0].$$

Property 6.7. *We have $\tau = [\tau_{\epsilon_1} \neq 0] \vee [\tau_{\epsilon_2} \neq 0]$, where τ_{ϵ_1} and τ_{ϵ_2} are the rational numbers given by*

$$\tau_{\epsilon_1} = \{M/2^{\epsilon_1}\} \qquad \text{and} \qquad \tau_{\epsilon_2} = \{\widetilde{M} \cdot 2^{-\epsilon_2 - k}\}.$$

Proof. The sticky bit collects the information ’being zero or not’ for all the bits that can have been discarded during the shifts. The first shift is by δ' and this information is contained in the last bit of M via $T_{\delta'}$. This last bit is either in τ_{ϵ_1} when $\epsilon_1 > 0$, or in τ_{ϵ_2} when $\epsilon_1 = 0$. The second shift if by $\epsilon_1 \geqslant 0$ and the information is contained in the ϵ_1 least significant bits of M. The third and last shift is by $\epsilon_2 + k$, which can be positive or nonnegative. □

Recall that $\widetilde{M}_H$ and $\widetilde{M}_L$ are respectively, the higher half and the lower half of the $2k$-bit unsigned integer $\widetilde{M}$. Then, from Property 6.7 we deduce that the sticky bit can be implemented as $\tau = [\mathrm{Tau} \neq 0]$ with

$$\mathrm{Tau} = T_1 \,|\, T_2,$$

where

$$T_1 = (M_L \ll 1) \ll \widetilde{\epsilon_1} \tag{6.19}$$

and

$$T_2 = \begin{cases} \Big(\widetilde{M}_H \ll (k - \epsilon_2)\Big) \,|\, \widetilde{M}_L & \text{if } \epsilon_2 > 0, \\ \Big(\widetilde{M} \ll (-\epsilon_2)\Big) \bmod 2^k & \text{if } \epsilon_2 \leqslant 0. \end{cases} \tag{6.20}$$

Like for the computation of $T_{\delta'}$, the straightforward implementation of T_1, such that $T_1 = M \ll (2k - \epsilon_1)$, may involve a shift by $2k$ bits. However, since ϵ_1 satisfies $0 \leqslant \epsilon_1 \leqslant p + 3 < k$, we can implement T_1 as in (6.19). For the binary32 format, we can reuse $\widetilde{\epsilon_1} = 31 - \epsilon_1$ from the computation of $\widetilde{M}$. The corresponding C code is at line 4 of Listing 6.5.

To compute T_2 for the case when $\epsilon_2 > 0$, according to (6.20), we have $k - \epsilon_2 < k$, and therefore the shift by $k - \epsilon_2$ bits is safe. The corresponding C code for the implementation of T_2 is shown at lines 13 to 16 of Listing 6.5.

When Tau is available, for $\circ = \mathrm{RN}$, the implementation of b is

$$b = g \wedge (\mathrm{Tau} \,|\, \ell_{p-1}),$$

where $g = G \,\&\, 1$, $\ell_{p-1} = G \,\&\, 2$, which is shown at line 20 of Listing 6.5.

For $\circ = \{\mathrm{RU}, \mathrm{RD}\}$, we need to compute $g \vee \tau = [\{\ell \cdot 2^{p-1}\} \neq 0]$. By using Theorem 6.2 and Property 6.7, we have $g \vee \tau = g \vee T_1 \vee T_2$ and we can implement the computation of $g \vee T_2$ as follows:

$$g \vee T_2 = \begin{cases} \big(\widetilde{M}_H \ll (k - \epsilon_2 - 1)\big) \,|\, \widetilde{M}_L, & \text{if } \epsilon_2 > 0, \\ \Big(\widetilde{M} \ll (-\epsilon_2 - 1)\Big) \bmod 2^k & \text{if } \epsilon_2 \leqslant 0. \end{cases}$$

Hence, for $\circ = \{\mathrm{RU}, \mathrm{RD}\}$, we can replace the implementation of T_2 at lines 13 and 16 of Listing 6.5 by

```
if(ep2 > 0) T2 = MtL | MtH << (31 - ep2);
else T2 = (uint32_t)(Mt << (-ep2 - 1));
```

Therefore, for $\circ = \mathrm{RU}$, we replace the computation of b at line 20 of Listing 6.5 by

```
b = (T1 | T2) && (S == 0);
```

Respectively, for $\circ = \mathrm{RD}$, we have

```
b = (T1 | T2) && S;
```

$\boxed{\text{Computing } s.}$ To implement the sign of the result, we can in fact reuse the if-else statement introduced for computing the integer M. Indeed, we have the following characterization:

Property 6.8. *The sign s of the result is such that* $s = \begin{cases} s_u \oplus \sigma & \textit{if } M_u < \lfloor M_v \cdot 2^{-\delta'} \rfloor, \\ s_u & \textit{otherwise.} \end{cases}$

Proof. Recall from (6.12) that $s = s_u \oplus s_m$. Then, for $\sigma = 0$ we have $s = s_u$ and the result is true. Assume now that $\sigma = 1$. Then $s_m = [m_u < m_v \cdot 2^{-\delta}]$. Since by Property 6.5 the condition $m_u < m_v \cdot 2^{-\delta}$ is equivalent to the condition $M_u < \lfloor M_v \cdot 2^{-\delta'} \rfloor$, we obtain

$$s = \begin{cases} s_u \oplus 1 & \text{if } M_u < \lfloor M_v \cdot 2^{-\delta'} \rfloor, \\ s_u & \text{otherwise,} \end{cases}$$

and the conclusion follows. □

Based on the above property, we implement $S = s \cdot 2^{31}$ for the binary32 format as shown at line 1 of Listing 6.5. On the ST231, we implemented $M_u < \lfloor M_v \cdot 2^{-\delta'} \rfloor$ by $[M_u < M'_v]$ by using Property 6.6.

$\boxed{\text{Computing } D \text{ and packing the result.}}$ For $D = d - e_{\min}$, we can in fact reuse the integer D' (implemented as $\max(D_u, 0)$) already computed for producing $\widetilde{M}$.

Indeed,

$$\begin{aligned} D &= \mu - \nu - e_{\min}, && \text{since } d = \mu - \nu, \\ &= D' - \nu, && \text{since } D' = \mu - e_{\min}. \end{aligned}$$

The corresponding C code is shown at line 18 of Listing 6.5.

When G is available, we have $L = G \gg 1$, and the final result for generic input is implemented as

$$R = (S \,|\, (D \ll (p-1))) + L + b.$$

The corresponding C code for the binary32 format is shown at line 21 of Listing 6.5.

Listing 6.5: Implementation of R for generic input (binary32 format and $\circ = \mathrm{RN}$).

```
0 Dp = max(Du, 0);    MH = M >> 32; ML = (uint32_t)M;
1                     if(Mu < Mvp) S = Su ^ sigma; else S = Su;
2 ep1 = min(Dp - Du, 27);
3 ep1t = 31 - ep1;
4 MtH = MH >> ep1;  T1 = (ML << 1) << ep1t;
```

```
MtL = ((MH << 1) << ep1t) | (ML >> ep1);
Mt = ((uint64)MtH << 32) + MtL;

lzMt = clzl(Mt);
lmbp = lzMt - 3;
nu = min(lmbp, Dp);
ep2 = 4 - nu;

if(ep2 > 0) {G = MtH >> ep2; T2 = MtL | MtH << (32 - ep2);}
else {Mt = ((uint64_t)MtH << 32) + MtL;
      Mtp = Mt << (-ep2);
      G = Mtp >> 32;  T2 = (uint32_t)Mtp;}
                      L = G >> 1;  g = G & 1;
Tau = T1 | T2;      D = Dp - nu;

b = g && (Tau | (G & 2));
Rgen = (S | (D << 23)) + L + b;
```

6.4 Detecting and handling special input

6.4.1 Detecting special input

Now that we have seen how to handle generic input, we turn to special input. We see first how to detect special input, that is, how to evaluation the condition C_{spec}. Recall from Equation (6.2) that we have

$$C_{\text{spec}} = C_{\text{nan}} \vee C_{\text{inf}} \vee C_{\text{ovf}} \vee C_{\text{zero}},$$

where $C_{\text{nan}} = [r \text{ is NaN}]$, $C_{\text{inf}} = [\text{one of } x,\, y,\, z,\, t \text{ is infinity}]$, $C_{\text{ovf}} = [\text{overflow occurs}]$, and $C_{\text{zero}} = [\rho \text{ is } \pm 0]$. Therefore, we can obtain C_{spec} from C_{nan}, C_{inf}, C_{ovf}, and C_{zero}, and here we present the implementation for each of these sub-conditions.

Before the discussion, we define some useful integers and conditions for the implementation of C_{spec} by using the standard encodings X, Y, Z, T of x, y, z, t as follows.

We compute the encoding of $\max(|x|, |y|)$ and check whether x or y is infinite:

$$\begin{aligned} M_1 &= \max(|X|, |Y|), \\ C_1 &= [M_1 = \iota(\infty)]. \end{aligned}$$

We compute the encoding of $\min(|x|, |y|)$ and check whether x or y is zero:

$$\begin{aligned} M_2 &= \min(|X|, |Y|), \\ C_2 &= [M_2 = 0]. \end{aligned}$$

Similarly, we have

$$\begin{aligned} M_3 &= \max(|Z|, |T|), \\ C_3 &= [M_3 = \iota(\infty)], \end{aligned}$$

and

$$\begin{aligned} M_4 &= \min(|Z|, |T|), \\ C_4 &= [M_4 = 0]. \end{aligned}$$

Remark that M_2, C_2, and M_4, C_4 are already used in the generic path to compute e_u.

Computing C_{nan}. Recall the three cases which lead to the result NaN for DP2. We have

$$C_{\text{nan}} = C_{\text{nan1}} \vee C_{\text{nan2}} \vee C_{\text{nan3}},$$

where

- $C_{\text{nan1}} = [\text{one of } x,\ y,\ z,\ t \text{ is NaN}]$,
- $C_{\text{nan2}} = [\text{either } xy \text{ or } zt \text{ is a product of the form } 0 \times \infty]$,
- $C_{\text{nan3}} = [xy \text{ and } zt \text{ are infinite with opposite signs}]$.

Using the integers M_i, C_i, $i = 1, \ldots, 4$ introduced just above, we can implement C_{nan1}, C_{nan2}, and C_{nan3} as follows:

- $C_{\text{nan1}} = [\max(M_1, M_3) > \iota(\infty)]$;
- $C_{\text{nan2}} = (C_1 \wedge C_2) \vee (C_3 \wedge C_4)$;
- $C_{\text{nan3}} = C_1 \wedge C_3 \wedge \sigma$ with $\sigma = s_{xy} \oplus s_{zt}$.

Computing C_{inf}, C_{ovf}, and C_{zero}. Implementing these three conditions is immediate as well:

- $C_{\text{inf}} = C_1 \vee C_3$;
- $C_{\text{ovf}} = [D \geqslant 2e_{\max}]$;
- $C_{\text{zero}} = [\operatorname{clz} \widetilde{M} = 2k]$.

Note that D and $\operatorname{clz} \widetilde{M}$ are already computed in the generic path.

6.4.2 Handling special input

As soon as special inputs are detected, we should handle them as specified in §6.2. The easiest case is when C_{nan} is true: we return qNaN independently of the rounding mode. Now, let us discuss how to handle other special input.

Handling C_{inf}. When C_{inf} is true, although the result is independent of the rounding mode as well, the sign of the result is that of the product introducing the infinity, which is either s_{xy} or s_{zt}. The C code is shown at lines 17 and 18 in Listing 6.6. If both xy and zt are infinite, when C_{nan} does not hold, we have $s_{xy} = s_{zt}$. Then, the selection at lines 17 and 18 is still correct.

Handling C_{ovf}. The result of overflow depends on the rounding mode, which is specified in (6.1). The C code for $\circ = \mathrm{RN}$ for the binary32 format is given at line 23 in Listing 6.6, where variable `S` is reused from the generic path.

For the other rounding modes (RU, RD, RZ), all we need to change is line 23. The corresponding C codes for the binary32 format are given below.

- Rounding up ($\circ = \mathrm{RU}$):

```
else if (Covf) return (S | 0x7f800000) - (S >> 31);
```

- Rounding down ($\circ = \mathrm{RD}$):

```
else if (Covf) return (S | 0x7f800000) - (S == 0);
```

- Rounding to zero ($\circ = \mathrm{RZ}$):

```
else if (Covf) return S | 0x7f7fffff;
```

Handling C_{zero}. When the exact result is zero, from the rules specified in §6.2, we can conclude that

- for $\circ = \mathrm{RD}$, we have $s = 1$;
- for other rounding modes, we have $s = s_{xy} \wedge s_{zt}$. Indeed, if the products xy and zt are nonzero, only operands with opposite signs can lead to the exact zero result, and therefore, $s_{xy} \,\&\, s_{zt} = 0$; if both xy and zt are zero, we still have $s_{xy} \,\&\, s_{zt} = 0$ when they have opposite signs, and we have $s_{xy} \wedge s_{zt} = s_{xy} = s_{zt}$ otherwise.

We give at line 24 in Listing 6.6 the C code for the handling the case of an exact result equal to zero. This code holds for $\circ = \mathrm{RN}$, but also for $\circ \in \{\mathrm{RU}, \mathrm{RZ}\}$. For $\circ = \mathrm{RD}$, we replace line 24 by

```
else if (Czero) return 0x80000000;
```

Listing 6.6: Detecting and handling special input of DP2 for the binary32 format, $\circ = \mathrm{RN}$

```
0 //handling generic input, detailed in Section 6.3
1
2 //computing M1,C1, and M3, C3
3
4 Cnan1 = maxu(M1, M3) > 0x7f800000;
5 Cnan2 = (C1 && C2) || (C3 && C4);
6 Cnan3 = C1 && C3 && sigma;
7 Cnan = Cnan1 || Cnan2 || Cnan3;
8
9 Cinf = C1 || C3;
```

```
Covf = D >= 254;
Czero = lzMt == 64;

Cspec = Cnan || Cinf || Covf || Czero;

if(C1) Sinf = Sxy;
else   Sinf = Szt; // Sxy, Szt computed in the generic path

if (Cspec){
  if (Cnan) return 0x7fc00000;
  else if (Cinf) return Sinf | 0x7f800000;
  else if (Covf) return S | 0x7f800000;
  else if (Czero) return Sxy & Szt;
}else{
  return Rgen;
}
```

6.5 Experimental results obtained on the ST231

6.5.1 Operator performances

Performances on the ST231. The latencies of DP2 for the binary32 format on the ST231 are given in the third column of Table 6.1. For comparison, the second column of the table displays the latencies of a naive implementation of DP2 (using two multiplications and one addition from FLIP 1.0). As shown in the fourth column, our fused operator achieves a speedup between 1.15 and 1.30, depending on the rounding mode. Since the speedups here do not account for the two function calls penalty, the exact performance gain obtained at run time can even be larger. Furthermore, recall that our DP2 operator provides the correctly-rounded result (only one rounding error) while the naive approach entails up to three rounding errors.

$\circ$	FLIP 1.0 $\circ(\circ(xy) + \circ(zt))$	DP2 $\circ(xy + zt)$	speedup
RN	68	55	1.24
RU	69	54	1.28
RD	69	53	1.30
RZ	59	51	1.15

Table 6.1: Latency comparison for DP2 and the naive implementation of $xy + zt$ using FLIP 1.0, for the binary32 format.

Table 6.2 shows that on the ST231 and for the binary32 format, the IPC of DP2 is extremely high: it ranges between 3.8 and 3.9 depending on the rounding modes. In fact, by studying the assembly codes, we observe that for $\circ =$ RD, all the bundles are fully occupied except the last one, and for other rounding modes, there are only three or four bundles which are not fully used. Therefore, our DP2 operator makes almost the best possible usage of the 4 issues of the ST231.

$\circ$	Latency L	Number N of instructions	IPC = N/L
RN	55	207	3.8
RU	54	206	3.8
RD	53	205	3.9
RZ	51	194	3.8

Table 6.2: Latency, code size, and IPC for DP2.

Comparison with fused multiply-add. The standard specified operator, fused multiply-add (FMA), which computes $\circ(xy + z)$, can of course be considered as a specialization of DP2. Assuming the same definition of generic input for FMA as for DP2, we have the same steps to compute FMA as those for DP2:

1. normalizing the input and preparing the two summands;
2. swapping the two summands to get u and v;
3. computing $\circ(u + v)$ using integer arithmetic.

Note that step 1 of FMA is slightly simpler than for DP2 since we only need to compute one product, namely xy, instead of two. However, step 2 and step 3 are exactly the same. Thus, we can derive an implementation of FMA directly from the one we have presented for DP2. Its performances are shown in Tables 6.3 and 6.4 below.

Table 6.3 gives the performances of FMA on the ST231. Since steps 2 and 3 are the most expensive parts of the implementation, the latency of FMA is not significantly smaller than that of DP2. For example, for $\circ$ = RN, FMA takes 46 cycles, while DP2 takes 55 cycles. Furthermore, since multiplication in FLIP 1.0 takes 21 cycles, we could get $\circ(\circ(xy) + zt)$ in $21 + 46 = 67$ cycles. This is more expensive than one call to DP2, which is both faster (55 cycles) and more accurate (only one rounding error).

The second column of Table 6.3 gives the latencies of a naive implementation of FMA (using one multiplication and one addition from FLIP 1.0). Here we see that the difference of latencies is only around one or two cycles depending on the rounding modes. For $\circ$ = RZ, the fused operator is even slower by one cycle. This is caused by the expensive normalization and rounding step of FMA. However, FMA should still be preferred to the naive approach , since it guarantees at most one rounding error instead of up to two.

Table 6.4 gives the IPCs of FMA for the binary32 format on the ST231, which range between 3.6 and 3.8. Although they are slightly smaller than for DP2, such IPC values still indicate a heavy use of the machine resources.

6.5.2 Application examples

FFT computation. Radix-2 butterfly is the basic computation element in performing FFT. As we have explained in the introduction part of the document (see §1.2.2),

$\circ$	FLIP 1.0 $\circ(\circ(xy) + z)$	FMA $\circ(xy + z)$	speedup
RN	47	46	1.02
RU	48	46	1.04
RD	48	45	1.07
RZ	41	42	0.98

Table 6.3: Latency comparison for FMA and the naive implementation of $xy + z$ using FLIP 1.0, for the binary32 format.

$\circ$	Latency L	Number N of instructions	IPC = N/L
RN	46	170	3.7
RU	46	167	3.6
RD	45	166	3.7
RZ	42	158	3.8

Table 6.4: Latency, code size, and IPC for FMA.

our production compiler can generate two DP2 and two addsub operators for each butterfly; here, we study the performance gain provided by our fast DP2 implementation on the ST231.

Listing 6.7 gives the typical butterfly implementation. We see that the variable `t1` at line 1 can be computed by three independent general operators (two multiplications and one subtraction) with three times rounding, or by one FMA and one multiplication, or by a single DP2 operator. Similarly, the variable `t2` can also be computed by several operators or by a single DP2.

Listing 6.7: Typical butterfly implementation.

```
0 for (k=j; k <= n; k=k+n2 ){
1  t1 = c*x[k+n1] - s*y[k+n1]; t2 = s*x[k+n1] + c*y[k+n1];
2  x[k+n1] = x[k] - t1;         x[k] = x[k] + t1;
3  y[k+n1] = y[k] - t2;         y[k] = y[k] + t2 ;
4 }
```

Table 6.5 gives the performance of running the FFT test suites from the UTDSP benchmark [Lee] on the ST231. The baseline is performing FFT by general addition and multiplication operators of FLIP 1.0. From column “DP2”, we see that using DP2 operators alone (without addsub operator) can almost be 1.2x faster than the baseline. Compared with the speedup introduced by FMA, selecting DP2 gives much more performance gain.

Furthermore, the computation at lines 2 and 3 of Listing 6.7 can be selected as two addsub operators (latencies shown in Table 1.1). The last column shows that to compute FFT by a conjunction of DP2 and addsub can be 1.59x faster than the baseline.

Graphics applications. Another application area where this approach is efficient is in software 3D-graphics pipeline, that operates in single precision mode.

Test suite	FLIP 1.0	FMA		DP2		DP2 and addsub	
FFT-256	324001	304545	[1.06]	271811	[1.19]	204194	[1.59]
FFT-1024	1609927	1512647	[1.06]	1352879	[1.19]	1010887	[1.59]

Table 6.5: Performances of FFT on the ST231 in # cycles and [speedups].

Table 6.6 gives the performance of running the whole `GeometryPipeline`, as well as the cost of important sub-computations of this pipeline:

- `Lerp` is a tri-linear interpolator;
- `BackFaceCull` computes the polygon visibility;
- `TransfromPt` is a 4D matrix by vector product computation.

The baseline is performing the test suites by using FLIP 1.0. The columns "FMA" and "DP2" give the latencies and speedups obtained when, respectively, either the FMA or the DP2 operator is available. The average speedup provided by DP2 is higher than that of FMA. The last column "FMA and DP2" gives the performances when both of these two fused operators are in use, and the speedups then range from 1.25 to 1.45.

Test sutie	FLIP 1.0	FMA		DP2		FMA and DP2	
GeometryPipeline	60803	53056	[1.15]	53555	[1.14]	48746	[1.25]
Lerp	23212	20523	[1.13]	15989	[1.45]	15989	[1.45]
BackFaceCull	9281	8214	[1.13]	7037	[1.32]	6903	[1.34]
TransformPt	17903	15303	[1.17]	15570	[1.15]	13903	[1.29]

Table 6.6: Performances of graphics applications on the ST231 in # cycles and [speedups].

Chapter 7

Simultaneous sine and cosine over a reduced range

Graphics and signal processing applications often require that sines and cosines be evaluated at a same floating-point argument, and in such cases a very fast computation of the pair of values is desirable. In this chapter, we study how to exploit the 32-bit VLIW integer architecture of the ST231 in order to perform this task accurately for IEEE single precision, including subnormals. We describe software implementations for sinf, cosf, and sincosf over $[-\pi/4, \pi/4]$ that have a proven 1-ulp accuracy and whose latency on the ST231 is 19, 18, and 19 cycles, respectively. Such performances are obtained by introducing a novel algorithm for simultaneous sine and cosine that combines univariate and bivariate polynomial evaluation schemes.

Most of the work in this chapter has been published in [JJL12].

7.1 Introduction

In this chapter we consider the sine and cosine functions, with special emphasis on their *simultaneous* computation, as in [Mar03]. Typically, the evaluation of any of these functions is decomposed into three steps [Mul06]:

- Range reduction, which computes $x^* \in [-\pi/4, \pi/4]$ and $k \in \mathbb{Z}$ such that $x^* = x - k\pi/2$;
- Evaluation of $\sin x^*$ or $\cos x^*$ depending on the value of $k \bmod 4$;
- Sign adaptation to reconstruct the result.

Clearly, range reduction can be shared by both functions, so that we focus here on the reduced range $[-\pi/4, \pi/4]$.

When multipliers are available on the target architecture, CORDIC-like shift-and-add methods need not be used and the computation of sine and/or cosine eventually often relies on the evaluation of one or several univariate polynomial approximations, be it in hardware [PKS00, Tis06, DdD07] or in software [Tan90, GB91, Mar00, CHT02, Rai06, LS07, crl, Shi10]. In particular, fixed-point univariate polynomials have been already employed for sine and cosine on ST231 [Rai06, §14], yielding respective latencies of 29 and 36 cycles, without accuracy guarantee.

In this chapter we propose a new approach based on the combination of fixed-point univariate and bivariate polynomials for more ILP exposure, along with rigorous accuracy analyzes. More precisely, the contributions of this chapter are as follows:

First, we introduce an algorithm for sine (resp. cosine) that uses a single bivariate (resp. univariate) polynomial approximation evaluated in a highly parallel fashion, and whose accuracy is shown to be $\leqslant 1$ ulp of the exact result.

Second, we deduce from these two algorithms a third one for simultaneous sine and cosine, whose hybrid univariate/bivariate nature brings high ILP exposure.

Third, we detail the corresponding C implementations, showing optimal schedules and very low latencies on a VLIW integer processor like the ST231: 19 cycles for `sinf`, 18 for `cosf`, and 19 for `sincosf`.

Thus, compared with [Rai06] speedups of $> 1.5\times$ for `sinf` and $2\times$ for `cosf` are obtained, together with proven 1-ulp accuracy. Also, `sincosf` yields both 1-ulp accurate sine and cosine in the same latency as it takes to compute sine alone.

As for all the other operators implemented during this thesis work, our codes have been optimized with the ST231 architecture in mind, but they are written in standard C and are thus portable. In addition, the design has been assisted by a software toolchain consisting of Sollya [CJL10] for certified polynomial approximants and of Gappa [Mel] and CGPE [MR11] to guarantee the accuracy and cost features of polynomial evaluations. Finally, our approach supports subnormals for free and can easily be adapted to other floating-point formats like double precision as soon as 64-bit integer arithmetic is available.

Throughout this chapter, u denotes the *unit roundoff* in precision $p = 24$, that is,

$$u = 2^{-24},$$

and $\mathbb{F}$ denotes the set of standard binary32 finite floating-point numbers.

Outline. The chapter is organized as follows. §7.2 introduces the 1-ulp accuracy specification of the evaluations for sine and cosine. §§7.3 and 7.4 present our algorithms and implementations, respectively; in each case we provide a theoretical analysis yielding a proven 1-ulp error bound, as well as implementation details and C codes.[6] Then, in §7.5 we describe our simultaneous computation of sine and cosine and analyze its performances on ST231.

7.2 Accuracy specification using the ulp function

Recall from §2.4 that given a floating-point system of precision p and minimal exponent $e_{\min}$, the ulp of any real number x is

$$\operatorname{ulp}(x) = \begin{cases} 0 & \text{if } x = 0, \\ 2^{\max\{e_{\min}, e\} - p + 1} & \text{otherwise,} \end{cases}$$

with e the unique integer power of two such that $2^e \leqslant |x| < 2^{e+1}$.

Thanks to the above definition, "1-ulp accuracy" means that our implementations conform to the following precise specification: assuming an input x in

$$I = \mathbb{F} \cap [0, \pi/4],$$

[6] The symmetry properties $\cos(-x) = \cos x$ and $\sin(-x) = -\sin x$ allow us to restrict our accuracy analyzes to nonnegative inputs, but the codes do handle the range $[-\frac{\pi}{4}, \frac{\pi}{4}]$.

for cosine we want an r in $\mathbb{F}$ such that

$$|r - \cos x| \leqslant \mathrm{ulp}(\cos x). \tag{7.1}$$

Similarly, for sine this means an r in $\mathbb{F}$ such that

$$|r - \sin x| \leqslant \mathrm{ulp}(\sin x). \tag{7.2}$$

Furthermore, for x zero we ensure in each case that r is the *exact* result: $r = 1$ for cosine and $r = \pm 0$ for sine.

7.3 Computing cosine

Over $[0, \pi/4]$ the cosine function is strictly decreasing and takes values between 1 and $1/\sqrt{2} \approx 0.707$. Thus, for $x \in I$, we have

$$\mathrm{ulp}(\cos x) = \begin{cases} 2u & \text{if } x = 0, \\ u & \text{otherwise,} \end{cases}$$

and to satisfy our accuracy requirement in (7.1) it suffices to ensure

$$|r - \cos x| \leqslant u. \tag{7.3}$$

7.3.1 Constant approximation when $x < 2^{-11}$

Of course, when x is "close enough" to zero then $\cos x$ will be "close enough" to one and can be approximated by a constant in $\mathbb{F}$. More precisely, by Taylor's theorem, for $x > 0$ there exists a real x_0 in $(0, x)$ such that

$$\cos x = 1 - \tfrac{\cos x_0}{2} x^2. \tag{7.4}$$

Hence $0 \leqslant 1 - \cos x \leqslant \frac{1}{2}x^2$ and the accuracy condition in (7.3) holds with $r = 1 - u$ as long as $x \leqslant 2\sqrt{u}$. For binary32 this threshold is equal to 2^{-11}.

Thus, in principle we would start by detecting whether x satisfies $x \leqslant 2^{-11}$ or not. However, we prefer to replace that condition by the *strict* inequality

$$x < 2^{-11},$$

which is just slightly stronger but has the advantage of being equivalent to deciding whether the biased exponent of x satisfies $E_x < e_{\max} - 11 = 116$. Since E_x will be needed anyway when handling inputs larger than 2^{-11} (as shall be seen in Section 7.3.2), this is a way of reusing it to filter out small inputs.

To sum up, when x is in $I_{<2^{-11}}$, that is, when

$$E_x < 116$$

then we simply proceed as follows: if $x > 0$ we return the encoding of $1 - u$, namely

$$127 \cdot 2^{23} - 1 = (3f7fffff)_{16};$$

if $x = 0$ we add 1 to this value in order to return the encoding of the exact result $\cos 0 = 1$.

Note that since $e_{\min} = -126$, this first subinterval $I_{<2^{-11}}$ contains zero as well as *all* positive subnormal numbers.

7.3.2 Polynomial approximation when $x \geqslant 2^{-11}$

Let us now see how to evaluate $\cos x$ when x is in $I_{\geqslant 2^{-11}}$. We start by proving that in this case any floating-point number r satisfying (7.3) has exponent equal to -1.

Lemma 7.1. *Let $x \in I_{\geqslant 2^{-11}}$ and $r \in \mathbb{F}$ be as in (7.3). Then*

$$\tfrac{1}{2} \leqslant r < 1.$$

Proof. By assumption, $\cos x - u \leqslant r \leqslant \cos x + u$. Since $0 \leqslant x \leqslant \pi/4$, we have $\cos x - u \geqslant \frac{1}{\sqrt{2}} - 2^{-p} \geqslant \frac{1}{2}$ as soon as $p \geqslant 3$, which is of course the case for binary32. Let us now check that $\cos x + u < 1$. Recalling that we have (7.4) with $x_0 < x$ and that the cosine function is decreasing on $[0, \pi/4]$, we obtain $\cos x_0 > \cos x$ and thus, for any x in I,

$$\cos x \leqslant \left(1 + \tfrac{x^2}{2}\right)^{-1}.$$

Now, this upper bound is less than $1 - u$ as soon as $x > \sqrt{2u/(1-u)}$, which is true since $x \geqslant 2^{-11} = 2\sqrt{u}$. □

Lemma 7.1 implies that our floating-point unknown $r = m_r \cdot 2^{e_r}$ is in fact a fixed-point number of the form

$$r = (0.1m_{r,1} \dots m_{r,23})_2. \tag{7.5}$$

To obtain r as in (7.3) and (7.5) we extend to cosine the approach introduced in [JKMR11] for square root and consisting in the truncation of a Q0.32 number v that approximates “accurately enough” from above the exact result. A rigorous sufficient condition on the accuracy of v is provided by the theorem below.

Theorem 7.1. *Let $x \in I_{\geqslant 2^{-11}}$, let $v = (0.v_1 v_2 \dots v_{32})_2$ be such that*

$$|v - g(x)| \leqslant \tfrac{u}{2} \quad \textit{with} \quad g(x) = \tfrac{u}{2} + \cos x,$$

and let r be the truncated value of v after 24 fraction bits. Then r is a binary32 number that satisfies (7.3).

Proof. We have $r = (0.v_1 v_2 \dots v_{24})_2$, which implies that r is a binary32 number and, since $u = 2^{-24}$, that

$$-u < r - v \leqslant 0.$$

On the other hand, v satisfies

$$0 \leqslant v - \cos x \leqslant u.$$

Hence $-u < r - \cos x \leqslant u$, from which (7.3) follows. □

To compute from x a value v as in Theorem 7.1 we essentially rely on polynomial approximation and evaluation. The process consists of three steps:

- Precompute a polynomial a that approximates g over $I_{\geqslant 2^{-11}}$ and whose fixed-point coefficients have at most 32 fraction bits. This approximation generates the error $\epsilon_0 = \max_{x \in I_{\geqslant 2^{-11}}} |a(x) - g(x)|$.

- Compute a 32-bit fixed-point approximation t to the floating-point input x. This approximation yields the error $\epsilon_1 = |a(t) - a(x)|$.
- Given t and the coefficients of a, compute an approximation v to $a(t)$ in 32-bit fixed-point arithmetic by choosing a suitable polynomial evaluation scheme. The associated evaluation error is $\epsilon_2 = |v - a(t)|$.

Thus, by the triangle inequality, the accuracy constraint $|v - g(x)| \leqslant \frac{u}{2}$ in Theorem 7.1 is satisfied as soon as we have

$$\epsilon_0 + \epsilon_1 + \epsilon_2 \leqslant \tfrac{u}{2}. \tag{7.6}$$

The next three paragraphs give some implementation details showing how to achieve the bound in (7.6) very efficiently on the ST231 architecture. A fourth paragraph details how to reconstruct the encoding of the result r directly from the encoding of v.

1. Precomputing the polynomial approximant a**.** A polynomial that satisfies the necessary condition $\epsilon_0 \leqslant \frac{u}{2}$ must have degree at least 6; this can be seen using the software tool Sollya [CJL10], which further gives us:

- coefficients of the form $a_0 = 1 + A_0 \cdot 2^{-32}$, $a_4 = A_4 \cdot 2^{-32}$ and $a_i = -A_i \cdot 2^{-32}$ for $i \in \{1, 2, 3, 5, 6\}$ and where the seven A_is are 32-bit unsigned integers. (Recall that on the ST231, these coefficients can be encoded directly in instruction words, thus avoiding using tables, whose access can be costly.)
- the rigorous bound $\epsilon_0 < 2^{-28.86}$.

2. Approximating x **by** t**.** For x in $I_{\geqslant 2^{-11}}$ we have

$$x = (\underbrace{0.00\ldots00}_{|e_x|\text{ zeros}}1m_{x,1}\ldots m_{x,23})_2, \quad -11 \leqslant e_x \leqslant -1,$$

showing that the fraction of x can be up to 34-bit long. An approximation of x that will be enough for our purpose is its truncation after 32 fraction bits, that is,

$$t = \lfloor x \cdot 2^{32} \rfloor / 2^{32} = (0.t_1 t_2 \ldots t_{32})_2.$$

Hence $0 \leqslant x - t < 2^{-32}$ and, using the software tool Gappa [Mel], we get the rigorous bound $\epsilon_1 < 2^{-32.98}$.

Furthermore, this approximation is stored into the 32-bit unsigned integer $T = t \cdot 2^{32}$, which can be deduced from the standard encoding X in (2.4) as follows: shift X left by 8 places, set the leftmost bit to 1, and then shift the result right by $|e_x| - 1 = 126 - E_x$ places. Lines 2 and 3 of Listing 7.1 give the corresponding C code, which on ST231 takes 4 cycles.

3. Evaluating $a(t)$ **fast and accurately in fixed point.** High ILP exposure is obtained by parenthesizing our degree-6 polynomial $a(y) = a_0 + \cdots + a_6 y^6$ as

$$\big((a_0 + a_1 y) + (a_2 + a_3 y)z\big) + \big((a_4 + a_5 y) + a_6 z\big)(z^2)$$

with $z = y^2$. With unbounded parallelism and latencies of 3 and 1 for × and +, this scheme takes 11 cycles, which is 2.18 times less than Horner's rule [Knu87, p. 486].

The corresponding C code using (unsigned) 32-bit integer arithmetic appears at lines 4 to 14 in Listing 7.1. (Similarly to squaring in §4.3, `mul` returns the 32 most significant bits of the exact product of two 32-bit unsigned integers.) We see that the parallelism constraints of the ST231 still allow for an 11-cycle latency. Concerning the accuracy of this scheme, we used Gappa to check that all the computed quantities are positive Q0.32 numbers and also to get a rigorous bound on $\epsilon_2 = |v - a(t)|$ with $v = V \cdot 2^{-32}$. In our case, it turns out that $\epsilon_2 < 2^{-30.04}$, so that overall (7.6) is satisfied.

4. Reconstructing the result. From Theorem 7.1 and since $V = v \cdot 2^{32}$, we conclude that the encoding of our result r is $R = 125 \cdot 2^{23} + \lfloor V/2^8 \rfloor$. Once V is available its computation thus takes 2 cycles, as line 16 of Listing 7.1 shows.

Listing 7.1: Cosine evaluation in binary32 over $[-\frac{\pi}{4}, \frac{\pi}{4}]$.

```
 0 Ex = (X >> 23) & 0xff;
 1 
 2 mx = (X << 8) | 0x80000000;
 3 T = mx >> (126 - Ex);
 4 Z = mul(T, T);         A5T = mul(A5, T);
 5 A1T = mul(A1, T);      A3T = mul(A3, T);
 6 
 7 Z2 = mul(Z, Z); A45 = A4 - A5T; A6Z = mul(A6, Z);
 8 A01 = A0 - A1T; A23 = A2 + A3T;
 9 A23Z = mul(A23, Z);
10 A46Z = A45 - A6Z;
11 A46 = mul(A46Z, Z2);
12 A03 = A01 - A23Z;
13 
14 V = A03 + A46;         // V encodes v = (0.v1...v32)
15 
16 R = (125 << 23) + (V >> 8);
17 if (Ex < 116) return 0x3f7fffff + ((X << 1) == 0);
18 else return R;
```

Remarks:

- When compiling the code in Listing 7.1 with st200cc, a latency of 18 cycles is achieved, thus indicating an optimal schedule: 4 cycles for `T`, 11 cycles for `V`, 2 cycles for `R`, and a final 1-cycle 'select' to choose between `R` and the constant `0x3f7fffff`.

- Also, it is not hard to see that the C code in Listing 7.1 works not only for the range $[0, \frac{\pi}{4}]$ but also for $[-\frac{\pi}{4}, \frac{\pi}{4}]$. This is solely due to the first line, so that for applications where x is known to be in $[0, \frac{\pi}{4}]$, one can replace this line by `Ex = X >> 23;` and thus reduce the latency by 1 cycle.

- The values of the polynomial coefficients A_i are displayed in Table 7.1.

A_0	`0x00000089`	A_1	`0x000003ef`
A_2	`0x7fffb575`	A_3	`0x0002110b`
A_4	`0x0ab188a0`	A_5	`0x000b29ea`
A_6	`0x005364d9`		

Table 7.1: Polynomial coefficients A_i.

7.4 Computing sine

Let us now consider the sine function $x \mapsto \sin x$. Over $[0, \pi/4]$ the sine function is strictly increasing and takes values between 0 and $1/\sqrt{2}$. This output range contains values that vary a lot in magnitude, making the ulp analysis more involved than for cosine. A classical workaround is to consider instead of $\sin x$ the function $\frac{\sin x}{x}$ whose range for $0 < x \leqslant \pi/4$ belongs to $[0.89, 1)$. Indeed, by Taylor's theorem there exists for $x > 0$ a real x_0 in $(0, x) \subset [0, \pi/4]$ such that

$$\sin x = x - \tfrac{\cos x_0}{6} x^3, \tag{7.7}$$

from which it follows that for any real x in $(0, \pi/4]$,

$$x(1 - \tfrac{x^2}{6}) < \sin x < x. \tag{7.8}$$

This enclosure implies that $\frac{\sin x}{x}$ ranges in $[1 - \epsilon, 1)$ with

$$\epsilon = \frac{\pi^2}{96} = 0.102\ldots.$$

It also serves as a basis for the following result, which describes the behavior of the ulp of sine and is a key ingredient for establishing the numerical accuracy of our implementations.

Lemma 7.2. *Let x be a real in $[0, \pi/4]$. Then*

$$\mathrm{ulp}(\sin x) = \begin{cases} \mathrm{ulp}(x) & \textit{if } x < 2^{e_{min}+1}, \\ \frac{1}{2}\mathrm{ulp}(x) \textit{ or } \mathrm{ulp}(x) & \textit{otherwise.} \end{cases}$$

Proof. If $x = 0$ then $\sin x = x$ and the result is true. If $0 < x < 2^{e_{\min}+1}$ then (7.8) gives $\sin x < x < 2^{e_{\min}+1}$ and by using (2.15e) we deduce that $\mathrm{ulp}(\sin x) = \mathrm{ulp}(x) = \alpha$. It remains to consider the case where $x \geqslant 2^{e_{\min}+1}$. In this case, we have $0 < x < 1$, so that (7.8) leads to

$$\tfrac{x}{2} < \sin x < x$$

and thus, using (2.15b), to $\mathrm{ulp}(\frac{x}{2}) \leqslant \mathrm{ulp}(\sin x) \leqslant \mathrm{ulp}(x)$. On the other hand, $x \geqslant 2^{e_{\min}+1}$ implies that both $\frac{x}{2}$ and x are in the normal range of $\mathbb{F}$, and (2.15c) then gives $\mathrm{ulp}(\frac{x}{2}) = \frac{1}{2}\mathrm{ulp}(x)$. Consequently, $\mathrm{ulp}(\sin x)$ ranges between $\frac{1}{2}\mathrm{ulp}(x)$ and $\mathrm{ulp}(x)$; but since ulps are integer powers of two, this means that $\mathrm{ulp}(\sin x)$ must be one of these two values. □

7.4.1 Approximation by x when $x < 2^{-11}$

Just like for cosine, for $x \in I$ in the neighborhood of zero we use the second-order Taylor approximation of $\sin x$ and thus return $r = x$; see (7.7). For binary32 it turns out that this choice guarantees the accuracy condition (7.2) as long as $x \leqslant 2^{-11}$. (This can be checked easily using arguments similar to those employed for the proof of Lemma 7.2.) Also, again like for cosine, in practice we work with the strict inequality

$$x < 2^{-11}$$

instead. Thus, the general structure of the C code for sine, shown in Listing 7.2, is similar to the one of Listing 7.1 and when $x \in I_{<2^{-11}}$ we simply return $r = x$.

7.4.2 Bivariate polynomial approximation when $x \geqslant 2^{-11}$

Assume now that our input x is in $I_{\geqslant 2^{-11}}$. Unlike for cosine and as can be expected, the exponent of the result is not anymore a constant known in advance. However, the result below shows that any floating-point number satisfying the accuracy constraint in (7.2) can have only two possible exponents, namely e_x or $e_x - 1$.

Lemma 7.3. *Let $x \in I_{\geqslant 2^{-11}}$ and $r \in \mathbb{F}$ be as in (7.2). Then, writing e_x for the exponent of x we have*

$$2^{e_x - 1} \leqslant r < 2^{e_x + 1}.$$

Proof. Since x is a normal number, we have $\mathrm{ulp}(x) = 2u \cdot 2^{e_x}$, so that (7.2) implies

$$\sin x - 2u \cdot 2^{e_x} \leqslant r \leqslant \sin x + 2u \cdot 2^{e_x}.$$

Furthermore, x is in $(0, \pi/4]$ and we can apply (7.8): the upper bound in (7.8) leads to $r < x + 2u \cdot 2^{e_x} = (m_x + 2u) \cdot 2^{e_x}$ and since $m_x \leqslant 2 - 2u$ we conclude that $r < 2^{e_x+1}$; using the lower bound gives $r > \big(m_x(1 - \frac{x^2}{6}) - 2u\big) \cdot 2^{e_x}$ and since $m_x \geqslant 1$ and $0 < x < 1$ we arrive at $r > (1 - \frac{1}{6} - 2u) \cdot 2^{e_x}$. For $p \geqslant 3$—which is the case for binary32—, this implies $r \geqslant 2^{e_x - 1}$ and the conclusion follows. □

Lemma 7.3 indicates that for floating-point solutions r to (7.2) the binary expansion of $r/2^{e_x}$ is either $(0.1 *_1 \cdots *_{23})_2$ or $(1. *_1 \cdots *_{23})_2$. In the theorem below we show how to recover such bits from truncating a suitable approximation v to a function of our input x.

Theorem 7.2. *Let $x \in I_{\geqslant 2^{-11}}$, let $v = (v_0.v_1 \ldots v_{31})_2$ be such that*

$$|v - h(x)| \leqslant \tfrac{u}{2} \quad \textit{with} \quad h(x) = \tfrac{u}{2} + \tfrac{\sin x}{2^{e_x}}$$

and with e_x the exponent of x, and let r be defined as

$$r = \begin{cases} (0.v_1 \ldots v_{23} v_{24})_2 \cdot 2^{e_x} & \textit{if } v < 1, \\ (1.v_1 \ldots v_{23})_2 \cdot 2^{e_x} & \textit{if } v \geqslant 1. \end{cases}$$

Then r is a binary32 number satisfying (7.2).

Proof. Note first that $e_x > e_{\min}$, which implies that r is a binary32 number and that $\mathrm{ulp}(x) = 2u \cdot 2^{e_x}$. Note also that the definition of h gives

$$0 \leqslant v \cdot 2^{e_x} - \sin x \leqslant u \cdot 2^{e_x}. \tag{7.9a}$$

If $v < 1$ then $r/2^{e_x}$ is the truncated value of v after 24 fraction bits, so that

$$-u \cdot 2^{e_x} < r - v \cdot 2^{e_x} \leqslant 0. \tag{7.9b}$$

Using (7.9a) and (7.9b) leads to the upper bound $|r - \sin x| \leqslant u \cdot 2^{e_x} = \frac{1}{2}\mathrm{ulp}(x)$, which by Lemma 7.2 is at most $\mathrm{ulp}(\sin x)$. Hence (7.2) is satisfied when $v < 1$.

Assume now that $v \geqslant 1$. In this case, $r/2^{e_x}$ is the truncated value of v after 23 fraction bits and thus

$$-2u \cdot 2^{e_x} < r - v \cdot 2^{e_x} \leqslant 0. \tag{7.9c}$$

We consider two subcases separately, depending on the value of the ulp of sine:

- If $\mathrm{ulp}(\sin x) = \mathrm{ulp}(x)$ then (7.2) is equivalent to $|r - \sin x| \leqslant 2u \cdot 2^{e_x}$, which holds thanks to (7.9a) and (7.9c).
- If $\mathrm{ulp}(\sin x) = \frac{1}{2}\mathrm{ulp}(x)$ then $\sin x < 2^{e_x} \leqslant x$, so that the upper bound in (7.9a) leads to
$$1 \leqslant v < 1 + u = (1.\underbrace{00\ldots00}_{23 \text{ zeros}}1)_2.$$
Since by assumption v is a Q1.31 number, this implies $v_1 = \cdots = v_{23} = 0$ and then $r = 2^{e_x}$. Therefore, in this subcase, (7.2) is equivalent to $2^{e_x} - \sin x \leqslant u \cdot 2^{e_x}$, which holds true thanks to (7.9a) and since $v \geqslant 1$.

Thus (7.2) holds for $v \geqslant 1$ too, which concludes the proof. □

Theorem 7.2 is obviously the counterpart of Theorem 7.1, that we have established for cosine in Section 7.3.2. However, its efficient implementation by means of polynomial approximation and evaluation is somehow more delicate, since the function h does not only depend on $\sin x$ but also on 2^{e_x}.

An efficient solution is to generalize to our context the bivariate polynomial approach of [JKMR11]: writing m_x for the significand of x, we have $\frac{\sin x}{2^{e_x}} = m_x \cdot \frac{\sin x}{x}$ and we can view $h(x)$ as the value at (m_x, x) of the bivariate function

$$\widetilde{h}(s, y) = \tfrac{u}{2} + s \cdot \tfrac{\sin y}{y}. \tag{7.10}$$

We then follow the same three-step process as seen for cosine, but with a *polynomial in two variables* instead of just one:

- Precompute a suitable bivariate polynomial $\widetilde{b}$ that approximates $\widetilde{h}$ over $[1, 2) \times I_{\geqslant 2^{-11}}$.
- In the evaluation pair (m_x, x), only x may not fit into 32 bits, so we approximate it by t as for cosine.
- Given m_x, t, and the coefficients of $\widetilde{b}$, compute an approximate value v to $\widetilde{b}(m_x, t)$.

This process generates three errors, say $\widetilde{\epsilon}_0, \widetilde{\epsilon}_1, \widetilde{\epsilon}_2$ (defined essentially as for cosine by replacing a and g with, respectively, $\widetilde{b}$ and $\widetilde{h}$). In order to apply Theorem 7.2 it suffices that the following analogue of (7.6) be satisfied:

$$\widetilde{\epsilon}_0 + \widetilde{\epsilon}_1 + \widetilde{\epsilon}_2 \leqslant \tfrac{u}{2}. \tag{7.11}$$

We conclude this section by detailing our implementation choices for those stages as well as for the reconstruction of the result.

1. Precomputing the bivariate polynomial approximant $\widetilde{b}$. Due to the shape of $\widetilde{h}$ in (7.10) we use a special bivariate polynomial of the form

$$\widetilde{b}(s, y) = \tfrac{u}{2} + s \cdot b(y).$$

Consequently,

$$\big|\widetilde{b}(s, y) - \widetilde{h}(s, y)\big| = s \cdot \Delta \quad \text{with} \quad \Delta = \big|b(y) - \tfrac{\sin y}{y}\big|,$$

and since $1 \leqslant s < 2$, we search for a polynomial b of minimal degree such that $\Delta \leqslant \frac{1}{2} \cdot \frac{u}{2} = 2^{-26}$. Using again Sollya, an even polynomial of degree 6 is found:

$$b(y) = b_0 + b_2 y^2 + b_4 y^4 + b_6 y^6,$$

where, for $0 \leqslant i \leqslant 3$, $b_{2i} = (-1)^i \cdot B_{2i} \cdot 2^{-32}$ and B_{2i} is a 32-bit unsigned integer. In addition, $\Delta < 2^{-27.84}$ and thus $\widetilde{\epsilon}_0 < 2^{-26.84}$.

2. Approximating x by t. Our C code for this step is the same as for cosine and appears at lines 2 and 3 of Listing 7.2. Again, we have $0 \leqslant x - t < 2^{-32}$ and using Gappa now leads to $|b(t) - b(x)| < 2^{-34.57}$ and thus to $\widetilde{\epsilon}_1 < 2^{-33.57}$.

3. Evaluating $\widetilde{b}(m_x, t)$ fast and accurately in fixed point. A highly parallel scheme for our bivariate polynomial is

$$\widetilde{b}(s, y) = \big((2^{-25} + sb_0) + (sb_2)z\big) + \big(sb_4 + (sb_6)z\big)(z^2),$$

where $z = y^2$. On ST231 this scheme takes 11 cycles and using the software tool CGPE [MR11], one can check that even with the knowledge that $s = m_x$ is available earlier than $y = t$ this latency cannot be reduced further by any other parenthesization.

A fixed-point implementation of this scheme is given in Listing 7.2: all the variables of the form `sB*` encode Q1.31 numbers, while as for cosine `Z` and `Z2` encode Q0.32 numbers. We have checked with Gappa the absence of overflow and that the Q1.31 number

$$v = V \cdot 2^{-31} = (v_0.v_1 \ldots v_{31})_2.$$

satisfies $\widetilde{\epsilon}_2 := |v - \widetilde{b}(m_x, t)| < 2^{-31.37}$, so that the accuracy constraint in (7.11) is eventually satisfied.

4. Reconstructing the result. By Lemma 7.3 and Theorem 7.2 the result is $r = (1.v_2 \ldots v_{24})_2 \cdot 2^{e_x - 1}$ if $v_0 = 0$, and $r = (1.v_1 \ldots v_{23})_2 \cdot 2^{e_x}$ if $v_0 = 1$. Therefore, its biased exponent is $E_r = E_x - 1 + v_0$ and its encoding has the form

$$R = H \cdot 2^{23} + L$$

with a high part $H = K + v_0$ such that $K = s_x \cdot 2^8 + E_x - 2$, and with a low part $L = \lfloor V/2^{7+v_0} \rfloor$.

For the cost, the bottleneck is the computation of v_0. Given V one could of course get it as `v0 = V >> 31;`. A faster way, shown at line 14 of Listing 7.2, consists in producing v_0 in parallel with the addition of `sB02` and `sB46` that gives V. Indeed, since `sB02` is available before `sB46`, we precompute the signed integer 2^{31} `- sB02` and compare it with `sB46` (whose first bit has been checked with Gappa to be always zero); this is equivalent to evaluating the condition $V \geqslant 2^{31}$, whose value is precisely v_0.

To summarize, this implementation brings R in 3 cycles after V. In addition, since K and H carry the sign bit of X, it works for signed inputs, that is, for any x in $\mathbb{F} \cap [-\frac{\pi}{4}, \frac{\pi}{4}]$.

Listing 7.2: Sine evaluation in binary32 over $[-\frac{\pi}{4}, \frac{\pi}{4}]$.

```
 0 Ex = (X >> 23) & 0xff;
 1
 2 mx = (X << 8) | 0x80000000;
 3 T = mx >> (126 - Ex);   sB6 = mul(mx, B6);
 4 Z = mul(T, T);          sB4 = mul(mx, B4);
 5                         sB2 = mul(mx, B2);
 6                         sB0 = mul(mx, B0);
 7 Z2 = mul(Z, Z);         sB6Z = mul(sB6, Z);
 8                         sB2Z = mul(sB2, Z);
 9                         sB0_up = 0x40 + sB0;
10 sB46Z = sB4 - sB6Z;
11 sB46 = mul(sB46Z, Z2);  sB02 = sB0_up - sB2Z;
12                         J = 0x80000000 - sB02;
13                         K = (X >> 23) - 2;
14 V = sB02 + sB46;        v0 = (int32_t)sB46 >= J;
15
16 H = (K + v0) << 23;     L = (V >> 7) >> v0;
17 R = H + L;
18 if (Ex < 116) return X; else return R;
```

Remarks:

- As for cosine, the st200cc compiler is able to optimally schedule the code in Listing 7.2, thus leading to a latency of $4 + 11 + 3 + 1 = 19$ cycles.

- One can restrict Listing 7.2 to the range $[0, \frac{\pi}{4}]$ by modifying only its first line (in exactly the same way as for cosine; see the remark at the end of Section 7.3.2). This will reduce the latency from 19 to 18 cycles.

- The values of the polynomial coefficients B_i are given in Table 7.2.

B_0	`0xfffffff2`	B_2	`0x2aaaa7e7`
B_4	`0x02220c05`	B_6	`0x000cc7d8`

Table 7.2: Polynomial coefficients B_{2i}.

7.5 Computing sine and cosine simultaneously

Given the codes for sine and cosine described and analyzed so far, an implementation of a 1-ulp accurate operator `sincosf` is straightforward: we essentially merge Listings 7.1 and 7.2, renaming some variables whenever necessary. The resulting C code has the form

```
uint64_t sincosf( uint32_t X ) {...}
```

and, given the encoding `X` of x in $\mathbb{F} \cap [-\frac{\pi}{4}, \frac{\pi}{4}]$, it returns `R` whose leftmost 32 bits contain the encoding of $r_1 \in \mathbb{F}$ such that $|r_1 - \sin x| \leqslant \mathrm{ulp}(\sin x)$, and whose rightmost 32 bits encode $r_2 \in \mathbb{F}$ such that $|r_2 - \cos x| \leqslant \mathrm{ulp}(\cos x)$.

The performances obtained by compiling the code for `sincosf` with the st200cc compiler (in -O3 and for the ST231 core) are summarized in Table 7.3. The results for the restricted range $[0, \frac{\pi}{4}]$ are indicated within square brackets. (Recall from §3.2.1 that thanks to if-conversion and the 'select' instruction, we get straight-line assembly code whose latency is independent of the value of the input.) Note that if we simply inline `sinf` and `cosf` and let the compiler merge the codes automatically, then st200cc achieves the same latency for `sincosf` as that in Table 7.3 but with one extra instruction.

	Latency L	Number N of instructions	IPC = N/L
sinf	19 [18]	31 [30]	1.6 [1.7]
cosf	18 [17]	28 [27]	1.4 [1.4]
sincosf	19 [18]	49 [48]	2.4 [2.5]

Table 7.3: Performances of 1-ulp accurate binary32 sine, cosine, and simultaneous sine and cosine on ST231.

The main conclusion is that we get both sine and cosine in exactly the same latency as it takes to compute sine alone.

Reasons for this are the relatively low IPC of separate sine and cosine, but also the fact that several instructions are common to both functions. Specifically, the number of instructions for `sincosf` is 10 less than the sum of those numbers for sine and cosine taken separately. On the other hand, by inspecting Listings 7.1 and 7.2 we see that sine and cosine share the computation of the biased exponent `Ex` and significand `mx` of the input, as well as the computation of `T`, `Z`, `Z2` and of the predicate `Ex < 116`.

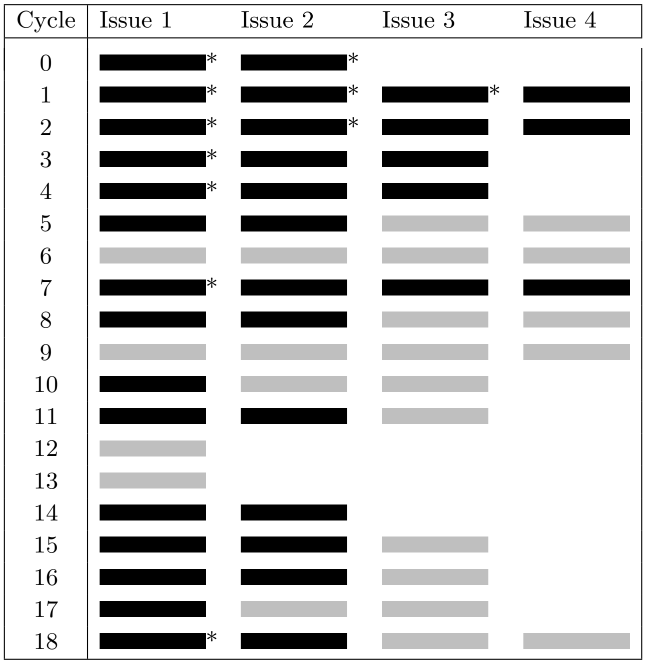
This said, as Table 7.4 shows, the two polynomial evaluations used within `sincosf` do not have much in common: they share only the computation of t^2 and t^4 (variables `Z` and `Z2`).

Figure 7.1 gives a precise description of the bundle occupancy when compiling `sincosf` with st200cc. The slots in black are those used to compute sine, those

	sine	cosine	shared by both	total
×	7	6	2	15
+, −	4	6	0	10
32-bit constants	4	6	0	10
9-bit constants	1	1	0	2

Table 7.4: Computational resources used by the two polynomial evaluations.

marked by $*$ are also used for cosine, and those in grey are used for cosine only. On the ST231, cosine can be fully computed in parallel with sine. In particular, since the latency of an integer multiplication here is 3 cycles, some computation for cosine can be conducted in bundles where no instructions for sine are executed, such as in cycles 6, 9, 12, and 13. On this target, every constant longer than 9 bits occupies one slot beyond the slot used by the instruction operating on it. Among the 19 bundles used, 15 are full or have 3 slots occupied. For this reason, the 4 issues of the ST231 are key to achieve a latency of 19 cycles.



Cycle	Issue 1	Issue 2	Issue 3	Issue 4
0	■*	■*		
1	■*	■*	■*	■
2	■*	■*	■	■
3	■*	■	■	
4	■*	■	■	
5	■	■	▨	▨
6	▨	▨	▨	▨
7	■*	■	■	■
8	■	■	▨	▨
9	▨	▨	▨	▨
10	■	▨	▨	
11	■	■	▨	
12	▨			
13	▨			
14	■	■		
15	■	■	▨	
16	■	■	▨	
17	■	▨	▨	
18	■*	■	▨	▨

Figure 7.1: Bundle occupancy for the `sincosf` operator on ST231.

Chapter 8

Compiler optimizations for floating-point support on the ST231

In this chapter, we focus on compilation aspects for the support of high-performance floating-point arithmetic on integer processors. First, to allow applications to benefit from custom floating-point operators, we study the selection of specialized, fused, and paired operators, that can be done at target-independent intermediate representation (WHIRL). Then, as the compiler is used to generate code for its own operators residing in a library (libgcc), we dedicate a specific effort at code generator intermediate representation (CGIR) level to ensure the best possible code selection when compiling the implementations of each floating-point operator on the ST231. Finally, for operators requiring to prove the positivity of some of their operands to be selected, such as fused square-add, we show how to augment the integer range analysis framework available at the CGIR level to detect this condition for floating-point variables. Compiling the UTDSP benchmark by the production compiler, we observe high usage of custom floating-point operators with speedups up to 1.59x.

8.1 Background

8.1.1 Intermediate representations for st200cc

ST200 VLIW compiler, st200cc, is organized as follows: the gcc-4.2.0 based front-end translates C/C++ source code into a first high level target independent representation called WHIRL, that is further lowered and optimized by the middle-end, including WHIRL global optimizer (WOPT), based on static single assignment form (SSA) representations, and optionally loop nest optimizer (LNO). It is then translated in a low-level target dependent representation, code generator intermediate representation (CGIR) for code generation, including code selection, low level loop transformations, if-conversion, scheduling, and register allocation.

The optimizations done to improve code selection when compiling library functions for floating-point support are done at the CGIR level. Most custom operators can be selected at the WHIRL level, except for the selection of fused square-add and non-negative add, which divert the integer range analysis optimization at CGIR level to analysis the range of floating-point variables.

8.1.2 Control on the selection of custom operators

Even though the FMA operator is described in the 2008 revision of the standard [IEE08, §5.4], the automatic reselection of FMAs in C11 application codes by compile-time recombination of addition and multiplication may lead to different application behavior, which may be an issue for example when strict bit-exactness is required to certify an application conformance to a standard. The selection of the DP2 operator, not even specified in the standard, may lead to similar difficulties.

Such options will as well allow us to measure the improvements introduced by the custom operators precisely.

Three options are designed for these two purposes:

- a specific option to disable the selection of FMA, `-mno-fused-madd`;
- a specific option to disable the selection of DP2, `-mno-fused-fp`;
- a specific option to enable the rebalancing transformation for DP2, `-ffast-math`.

The rebalancing transformation refers to the re-association of additions and multiplications of an n-dimensional product to have maximum number of DP2 operators selected, see §8.2.3.

Organization of the rest of the chapter. First, we show in §8.2 how to select squaring, scaling, two-dimensional dot products, and paired operators (addsub and sincos) at WHIRL level. Then, §8.3 details the optimizations that we introduce at CGIR level, which includes various peephole optimizations for code selection and the enhancement on the integer range analysis framework to support floating-point variables. Finally, in §8.4 we provide the experimental result of our production compiler for UTDSP benchmark's FFT test suite.

8.2 Selection of custom floating-point operators at WHIRL

In this section, we first introduce the target-independent intermediate presentation, named WHIRL, and we describe how the selection of custom floating-point operators can be carried out at this level, namely for squaring, scaling by a constant, two-dimensional dot product, and paired operators.

Working at this level has several benefits: the transformations done are target- and language- independent, and are relatively easy to implement in the existing compiler framework.

8.2.1 WHIRL intermediate representation

The compiler optimization components are driven to operate on the various levels of the target- and language-independent WHIRL intermediate representation [Ope00a], created by a specific bridge with a GNU gcc/g++ front-end.

Five levels of WHIRL are defined: very high (VH), high (H), Mid (M), low (L), and very low (VL) WHIRL. Each optimization phase is defined to work at a specific level of WHIRL and Figure 1 in [Ope00a] presents optimizations proceeding together with the process of continuous lowering from one WHIRL level to the next lower level. At the end, the code generator translates the lowest level of WHIRL into its own internal representation that matches the target machine instructions. WHIRL thus serves as the common IR interface among all back end components. A WHIRL file generated by the front-end consists of *WHIRL instructions* and *WHIRL symbol tables*. Now, we will give a general view of these two components.

WHIRL instructions. The instruction part of the WHIRL file represents the program code, organized in program units (PUs). The WHIRL instructions are linked in strictly tree form, and we refer to each node in the tree as a WHIRL node. Table 8.1 presents the layout of a WHIRL node.

field	description	API
prev	previous pointer	`WN* WN_prev(const WN*)`
next	next pointer	`WN* WN_next(const WN*)`
linenum	source position information	`uint64 WN_linenum(const WN*)`

offset (union)	Offsets for memory operators, label number, flags for calls, pragmas, trip count for loop_info, element size for array, etc.	`int32 WN_load_offset(const WN*)`
		`int32 WN_lda_offset(const WN*)`
		`int32 WN_store_offset(const WN*)`
		`int16 WN_loop_trip_est(const WN*)`
		`int64 WN_element_size(const WN*)`
		...

st_idx (union)	symbol table index, type index, flags for loop_info, etc.	`ST_IDX WN_st_idx(const WN*)`
		`TY_IDX WN_ty (const WN*)`

operator	WHIRL operator	`OPERATOR WN_operator(const WN*)`
rtype	result type	`TYPE_ID WN_rtype(const WN*)`

kid_count	number of kids for n-ary operators, or bit offset/size for bit manipulation operators	`int WN_kid_count(const WN*)`
		`uint WN_field_id(const WN*)`
		`uint WN_bitsize/offset(const WN*)`

desc	Operands type	`TYPE_ID WN_desc(const WN*)`
map_id	index into map_table	`int32 WN_map_id(const WN*)`

parameters (union)	parameters for kids, const_val, block list, pragma, etc.	`WN *WN_kid[0,1,2,3]`
		`WN_const_val`

Table 8.1: Layout of a WHIRL node.

A WHIRL opcode (prefixed with `OPC_`) is specified by three fields: *operator*, *desc*, and *rtype*.

WHIRL node instruction semantic defined by an operator is started with prefix `OPR_`. Some common operators are as follows:

- Structured Control Flow, such as `DO_LOOP`, `IF`, `FUNC`.
- Statements, such as `CALL`, `ASM`.
- Expressions, such as `ADD`, `NEG`.

- Leaves, such as `LABEL`, `CONST`.

Predefined data types are attached to WHIRL nodes using `MTYPE`, which stands for machine type. `MTYPE`s are used for the fields *desc* and *rtype*, which respectively specify the types of operand and result. Examples of predefined `MTYPE`s:

flag	description
I1, I2, I4, I8	8, 16, 32, 64-bit signed integer
U1, U2, U4, U8	8, 16, 32, 64-bit unsigned integer
F4, F8	binary32, binary64 floating-point number
C4, C8	binary32, binary64 floating-point complex number
V	void
B	boolean

Table 8.2: Examples of predefined `MTYPE`s.

Examples of WHIRL opcodes:

- `OPC_F4F4LDID` = `OPR_LDID` + `RTYPE(MTYPE_F4)` + `DESC(MTYPE_F4)`.
- `OPC_F4MPY` = `OPR_MPY` + `RTYPE(MTYPE_F4)` + `DESC(MTYPE_V)`;

As we see in the second example, the type of the operands for a multiplication is void. This is because the two operands of the multiplication are two independent WHIRL nodes linked as kid nodes to the multiplication node (more details shown in Listing 8.2). Thus, from the view of the opcode of this WHIRL node, the operands are void type.

WHIRL symbol tables. The WHIRL symbol table is made up of a series of tables. They are designed for compilation, optimization and storage efficiency. The way the tables are organized closely corresponds to the compiler's view of the symbol table. The model also enhances locality in references to the tables [Ope00b].

As symbol tables are not really important during the implementation of our optimizations, we do not provide more details on them in this document.

Example of WHIRL at different levels. A program unit is represented as a distinct tree, starting from a `FUNC_ENTRY` node. To compile the C code in Listing 8.1, the floating-point multiplication is represented by WHIRL opcode `OPC_F4MPY` at H WHIRL, shown in Line 13 in Listing 8.2. However, as floating-point arithmetic is supported by library functions, the opcode `OPC_F4MPY` is lowered to intrinsic opcode at M WHIRL, shown in Listing 8.3, and finally lowered to function call at L WHIRL, shown in Listing 8.4.

Listing 8.1: C code for the example for WHIRL presentations.

```
float mul(float x, float y){
  return x*y;
}
```

Listing 8.2: Example of H level WHIRL presentation

```
FUNC_ENTRY <level:1,idx:23,name:mul>
 IDNAME ofst:0 <level:2,idx:1,name:x> T<id:13,name:.predef_F4,align:4>
 IDNAME ofst:0 <level:2,idx:2,name:y> T<id:13,name:.predef_F4,align:4>
BODY
 BLOCK
 END_BLOCK
 BLOCK
 END_BLOCK
 BLOCK
 PRAGMA 0 120 <null-st> val:0 (0x0) # PREAMBLE_END
 LOC 1 2   return x*y;
   F4F4LDID ofst:0 <level:2,idx:1,name:x> T<id:13,name:.predef_F4,align:4>
   F4F4LDID ofst:0 <level:2,idx:2,name:y> T<id:13,name:.predef_F4,align:4>
  F4MPY
 F4RETURN_VAL
 END_BLOCK
```

Listing 8.3: Example of M level WHIRL presentation

```
F4INTRINSIC_OP 2 <961,MULS> 0
```

Listing 8.4: Example of L level WHIRL presentation

```
F4CALL 64 <level:1,idx:25,name:__muls> T<id:30,name:,align:1> # flags 0x40
```

8.2.2 Squaring and scaling by a constant

Squaring and scaling by a constant are specialized operators of general multiplication. In the production compiler, a floating-point multiplication can be selected as a square, when its two operands are identical. Meanwhile, a floating-point multiplication can be selected as a scaling operator, when one of its operands is a constant of powers of 2. Tabel 8.3 gives the patterns that can be selected as squaring or scaling. Here x is a C expression, which can be a constant, a single variable or the statement of operators acting on operands.

Expression	C type of expression x	Function selected in the ST231 `libgcc`
$x \cdot x$	`float`	`__squares`
	`double`	`__squared`
$x \cdot 2$, $x + x$	`float`	`__mul2s`
	`double`	`__mul2d`
$x \cdot (-2)$	`float`	`__nmul2s`
	`double`	`__nmul2d`
$x/2$, $x \cdot 0.5$	`float`	`__div2d`
	`double`	`__div2d`
$x/(-2)$, $x \cdot (-0.5)$	`float`	`__ndiv2s`
	`double`	`__ndiv2s`
$x \cdot C$, $C = 2^n$, $n \neq \pm 1$	`float`	`__scalbns`
	`double`	`__scalbnd`
$x \cdot C$, $C = -2^n$, $n \neq \pm 1$	`float`	`__nscalbns`
	`double`	`__nscalbnd`

Table 8.3: Pattern-matching table for Squaring and Scaling by a constant.

Selecting squaring and scaling. They are selected during lowering H WHIRL (opcode = `OPC_F4MPY`) to M WHIRL (opcode = `OPC_F4INTRINSIC_OP`). Now, we explain how to select squaring and scaling by computing the intrinsic ID by the ST231-dependent function `WN_To_INTRINSIC` with the information of the operands (`WN* kids[]`).

First, at target-independent level, we get the information of the operands, and then call `WN_To_INTRINSIC` to compute the intrinsic ID.

```
kids[0] = RT_LOWER_expr(WN_kid0(tree));
kids[1] = RT_LOWER_expr(WN_kid1(tree));
intrinsic = WN_To_INTRINSIC(opcode, kids);
```

- Squaring. In `WN_To_INTRINSIC`, check if the two operands are identical by comparing the their WHIRL nodes for squaring.

```
if (opcode == OPC_F4MPY){
  id = INTRN_MULS;
  if (WN_Compare_Trees(kids[0],kids[1]) == 0) {
    id = INTRN_SQUARES;
  }else{
    //selecting scaling ...
  }
}
```

- Scaling. Scaling by a constant, such as mul2, div2, is also selected during lowering high WHIRL to Mid WHIRL by checking the constant of a multiplication when computing the intrinsic ID. Meanwhile, mul2 can also be selected from addition (x + x), which is similar to the selection of square. For every floating-point addition, we check the identity of the two operands by function WN_Compare_Trees.

```
{
  TCON tc = Const_Val(kids[1]);
  if(Targ_Is_Power_Of_Two(tc)){
    UINT32 factor =  TCON_uval(tc);
    INT32 exp = (factor >> 23)&0xff;
    UINT32 sign = factor >> 31;
    UINT32 mantissa = factor << 9;
    if(exp == 0){
      UINT32 nz = countLeadingZeros(mantissa); // nz in [0,22]
      exp = -nz - 127;
    }else
      exp = exp - 127;
    switch(exp){
      case 1:
          if(sign) id = INTRN_NMUL2S;
          else     id = INTRN_MUL2S;
          break;
      case -1:
          if(sign) id = INTRN_NDIV2S;
          else     id = INTRN_DIV2S;
          break;
      default:
          // scalb(float x, int n), computing (+/-)2^n*x, detected.
          // n = exp.
          kids[1] = WN_Intconst(MTYPE_I4,exp); //create a WN tree for n
              and point kids[1] to it.
          if(sign) id = INTRN_NSCALBS;
          else     id = INTRN_SCALBS;
          break;
    }
  }
}
```

Creating the M WHIRL node for multiplication and its custom operators. With the obtained intrinsic ID, intrinsic, we create the new M WHIRL tree for the multiplication. Although for operations like squaring, mul2, and div2, we need only one operand, we still have to lower the two operands because the creation of M WHIRL node is on the target-independent level, and we can not check the IDs.

```
nd = Extension_Aware_CreateIntrinsic(OPCODE_make_op(OPR_INTRINSIC_OP, res,
    MTYPE_V), intrinsic, 2, kids);
```

8.2.3 Two-dimensional dot products and sums of squares

In this section, we will first present the basic idea to select two-dimensional dot products (DP2), and then we will see how to improve the selection when the expression is an n-dimensional dot product, which is normally lowered to only one DP2 and n−2 FMA, as by default, the compiler lowers the expressions from left to right. In the end, we will also explain the selection of sums of squares (SOS), which is considered as a specialized operator of DP2.

Selecting DP2. Three kinds of patterns can be selected as DP2. According to the type of expressions x, y, z, and t, expression $x \cdot y + z \cdot t$ is selected as __dps(x, y, z, t), which means single precision for dot product, or respectively, __dpd(x, y, z, t) for double precision. Similarly, $x{\cdot}y{-}z{\cdot}t$ is selected as __dpsubs(x, y, z, t) or __dpsubd(x, y, z, t), and $-x \cdot y - z \cdot t$ is selected as __ndps(x, y, z, t) or __ndpd(x, y, z, t). Table 8.4 summarizes the selections as follows.

Expression	C type of expressions x, y, z, and t	Function selected in the ST231 libgcc
$x \cdot y + z \cdot t$	float	__dps
	double	__dpd
$x \cdot y - z \cdot t$	float	__dpsubs
	double	__dpsubd
$-x \cdot y - z \cdot t$	float	__ndps
	double	__ndpd

Table 8.4: Pattern-matching table for DP2.

DP2 operators are selected at H WHIRL, and the selections are made out of the opcodes such as OPC_F4ADD, OPC_F4SUB, and OPC_F4MPY, with the rules shown in Figure 8.1. In this figure, we give the bottom-up view of the WHIRL trees that can be selected as DP2 operators. The right-justified opcodes are the kid nodes of the opcode below. Although we give only the rules for single precision here, the rules are exactly the same for double precision, for which we only need to replace F4, the rtype field of the WHIRL node, by F8.

Now we check every addition and subtraction to collect DP2. The following three listings show the selection of two-dimensional dot products. As explained in §8.1.2, option -mno-fused-fp is used to control the selection of DP2, this control is checked in the function WN_DP_Allowed in the following three listings.

- checking addition:

Listing 8.5: Selecting OPC_F4(F8)DP from addition.

```
case OPR_ADD:
    TYPE_ID  type = WN_rtype(tree);
    WN  *l= WN_kid0(tree);
    WN  *r= WN_kid1(tree);
    if(WN_DP_Allowed(type) && WN_operator_is(l, OPR_MPY) &&
        WN_operator_is(r, OPR_MPY))
        return_wn = WN_DP(type, WN_kid0(l), WN_kid1(l),WN_kid0(r),
            WN_kid1(r));
    break;
```

- checking subtraction:

Listing 8.6: Selecting OPC_F4(F8)DPSUB from subtraction.

```
case OPR_SUB:
    TYPE_ID  type = WN_rtype(tree);
    WN  *l= WN_kid0(tree);
    WN  *r= WN_kid1(tree);
    if(WN_DP_Allowed(type) && WN_operator_is(l, OPR_MPY) &&
        WN_operator_is(r, OPR_MPY))
        return_wn = WN_DPSub(type, WN_kid0(l), WN_kid1(l),WN_kid0(r),
            WN_kid1(r));
    break;
```

- checking `NDP`:

Listing 8.7: Selecting OPC_F4(F8)NDP.

```
case OPR_NEG:
    case OPR_DP:
        TYPE_ID  type = WN_rtype(tree);
        return_wn = WN_NDP(type, WN_kid0(child), WN_kid1(child), WN_kid2(
            child), WN_kid(child,3));
        break;
      ...
    break;
```

```
  OPC_F4MPY
  OPC_F4MPY
OPC_F4ADD     →  OPC_F4DP


  OPC_F4MPY
  OPC_F4MPY
OPC_F4SUB     →  OPC_F4DPSUB


  OPC_F4DP
OPC_F4NEG     →  OPC_F4NDP
```

Figure 8.1: Rules to select DP2 for single precision.

Selecting DP2 for n-dimensional dot products. An n-dimensional dot product, a chain of multiplications and additions, is normally lowered to only one DP2 and n−2 FMA, as by default, the compiler lowers the expressions from left to right. For example, a four-dimensional dot product, $a_0 \cdot a_1 + a_2 \cdot a_3 + a_4 \cdot a_5 + a_6 \cdot a_7$, is lowered to

$$\text{FMA(FMA(DP2}(a_0, a_1, a_2, a_3), a_4, a_5), a_6, a_7).$$

However, a faster way to evaluate this expression on the ST231 can be

$$\text{ADD(DP2}(a_0, a_1, a_2, a_3),\ \text{DP2}(a_4, a_5, a_6, a_7)).$$

We realize this transformation by splitting the FMA trees which have at least 2 FMAs nested. The C implementation is in Appendix B.

Since this transformation changes the precision of the result, the compilation option `-ffast-math` is used to control this optimization.

Expression	C type of expressions x, and z	Function selected in the ST231 `libgcc`
$x \cdot x + z \cdot z$	`float`	`__soss`
	`double`	`__sosd`

Table 8.5: Pattern-matching table for SOS.

Selecting sum of squares. When lowering DP2 from H to M WHIRL (intrinsic opcode), we can check the identity of the operands of the two multiplications to select sum of squares shown in Listing 8.8, which is very similar to the selection of square.

Listing 8.8: Computing the intrinsic ID for SOS.

```
if(opcode == OPC_F4DP){
  if ((WN_Compare_Trees(kids[0],kids[1]) == 0) && (WN_Compare_Trees(kids[2],
      kids[3]) == 0))
  id = INTRN_SOSS;
}
```

Creating M WHIRL node. Listing 8.9 shows how to create the M WHIRL node. Although we can not check the intrinsic ID at the target-independent level, as we have only one specialized operator for DP2 till now, we can determine the number of operands to lower by a simple comparison of whether the ID we obtain equals that of the general one. As shown in Listing 8.9, `intrinsic_general` in Line 0 contains the intrinsic ID of a DP2 and `intrinsic` in the same line is what we obtain from function `WN_To_INTRINSIC`. Therefore, only two operands are lowered for SOS and all the four operands must be lowered otherwise.

Listing 8.9: Creating M level WHIRL for DP2 or SOS.

```
0 if((intrinsic_general!=intrinsic)&& (type==MTYPE_F4))
```

```
  nd = Extension_Aware_CreateIntrinsic(OPCODE_make_op(OPR_INTRINSIC_OP, res,
      MTYPE_V), intrinsic, 2, kids);
else
  nd = Extension_Aware_CreateIntrinsic(OPCODE_make_op(OPR_INTRINSIC_OP, res,
      MTYPE_V), intrinsic, 4, kids);
```

8.2.4 Paired operators

Expression	C type of expressions x, and y	Function selected in the ST231 `libgcc`
$x + y$, $x - y$	`float` `double`	`__adspairs` `__adspaird`
$\sin x$, $\cos x$	`float` `double`	`__sincoss` `__sincosd`

Table 8.6: Pattern-matching table for paired operators.

Here we detail only the selection for addsub pair, but it is exactly the same method to select sincos pair. The compiler replaces all additions and subtractions by a call to a specific function returning a pair of floats. As addition and subtraction are 'pure' functions known by the compiler not to have any side-effect, when both results are required for the same input, the specific function will be called only once after redundancy elimination.

Indeed, the selection of paired operators is derived from the lowering process of complex numbers. At H level, we replace a single addition (or subtraction) by a paired operator with return type as for single-precision complex numbers (`MTYPE_C4`), shown in Listing 8.10. Listing 8.11 displays that an addition is lowered as the real part of a complex number and a subtraction is lowered as the imaginary part. Then, we create a new WHIRL node with opcode for paired operators.

Listing 8.10: Setting return type for paired operator.

```
switch (rtype)
    case MTYPE_F4: new_rty=MTYPE_C4; //treat the pair as complex data
         break;
```

Listing 8.11: Lowering additon and subtraction as complex data.

```
switch ( OPCODE_operator(old_wn_opc) ) {
  case OPR_ADD:
      addsub_part = OPCODE_make_op(OPR_REALPART,rtype,desc);
      break;
  case OPR_SUB:
      addsub_part = OPCODE_make_op(OPR_IMAGPART,rtype,desc);
      break;
      ...
}
const OPCODE addsub_opc = OPCODE_make_op(OPR_ADDSUB,new_rty,desc);
WN *addsub = WN_CreateExp2(addsub_opc, WN_kid0(old_wn),WN_kid1(old_wn));
WN *part = WN_CreateExp1(addsub_part, addsub);
*new_wn = part;
```

If no paired operators is selected, we will decompose it as shown in Listing 8.12.

Listing 8.12: Decomposing the paired operator.

```
switch (OPCODE_rtype(kid0_opc)){
  case MTYPE_C4: rtype =  MTYPE_F4; break;
}
const OPCODE add_or_sub_opc = old_wn_opr == OPR_REALPART ?
      OPCODE_make_op(OPR_ADD, rtype,OPCODE_desc(kid0_opc) ) :
      OPCODE_make_op(OPR_SUB, rtype,OPCODE_desc(kid0_opc) );

WN *add_or_sub = WN_CreateExp2(add_or_sub_opc, WN_kid0(kid0),WN_kid1(kid0));
*new_wn = add_or_sub;
```

If both results are required, the paired operator is called only once after redundancy elimination. At M level, it is lowered to intrinsic as shown in Listing 8.13.

Listing 8.13: Lowering paired operator to intrinsic opcode.

```
case OPR_ADDSUB:
    {
      TYPE_ID  rtype = OPCODE_rtype (RT_LOWER_opcode(tree));
      INTRINSIC mid;
      OPCODE new_intr_opc;
      switch (rtype){
        case MTYPE_C4:
            mid = OPCODE_To_INTRINSIC(WN_opcode(tree));
            new_intr_opc = OPC_C4INTRINSIC_OP;
            break;
      }

      //lowering the operands here
      ...

      //create the new WHIRL with intrinsic opcode
      nd = WN_Create_Intrinsic(new_intr_opc,mid, 2, kids);
      break;
    }
```

8.3 Various optimizations at CGIR

In this section we present our work on the Code Generator Intermediate Representation (CGIR), namely:

- peephole-like optimizations to ensure the best possible code generation for the implementations of floating-point operators;
- the extension of the integer range analysis framework for floating-point specialization.

Contrary to the transformations done at the higher WHIRL level, the optimizations done here are target-specific but remain language-independent.

The motivation for working at this lower CGIR level is twofold. First, most of the high- and mid-level compiler optimizations have been carried out, exposing precise properties of the program that can be approximated by the Range Analysis framework. Second, the semantic of each CGIR operator is precisely known for the target, enabling even more precise Range Analysis computations.

These properties are important to implement peephole-like optimizations that would be otherwise impossible to implement without range information, and to extend the Integer Range Analysis framework to prove positivity of floating-point arguments.

8.3.1 CGIR overview

The lowest WHIRL level is finally lowered to a target dependent intermediate representation, CGIR. Several target dependent optimizations can be carried out at CGIR, including low level loop transformations, if-conversion, scheduling, register allocation, and code selection. The integer range analysis framework is implemented in the extended block optimizer (EBO), which works on extended blocks and performs forward propagation, common expression elimination, constant folding, dead code elimination, and special case transformations that are specific to an architecture.

Basic structures of CGIR. The CGIR implementation is based on three kinds of data structures: basic block, operation, and temporary name.

Basic block. Intermediate codes are partitioned into basic blocks (BB) [ALSU06], which are maximal sequence of consecutive three-address instructions with the properties that:

- the control flow can only enter the basic block through the first instruction of the block, that is, there are no jumps into the middle of the block;
- control will leave the block without halting or branching, except possibly at the last instruction in the block.

The data structure of basic blocks in st200cc is shown in Table 8.7

CGIR operation. For each WHIRL node, one or a few CGIR operations (OPs) are generated and inserted into a basic block. The structure of OP in st200cc is shown

structure member	description	
`id`	unique ID	general information
`flags`,	`BB_entry, BB_scheduled, BB_asm,` etc	
`annotations`	list of annotations (`label, pragma, loopinfo,`, etc)	
`rid`	region from WHIRL that contains this BB	
`next, prev`	sequential ordering	control-flow information
`preds, succs`	control-flow arcs with probabilities	
`freq`	estimated or profiled execution frequency	
`loop_head_bb`	basic block head of the loop that contains this BB	loops information
`unrollings`	number of times a BB has been unrolled	
`nest_level`	number of times a bb has been unrolled	
`ops`	sequential list of operations	operations information
`next_op_map_idx`	internal field to give a unique ID for operations in a BB	
`bb_regs`	live information	
`branch_wn`	information from the WHIRL on the branch instruction at the end of a BB	

Table 8.7: Structure of BB in st200cc.

in Table 8.8. Each OP represents a unique target machine instruction defined by its *opr* opcode. OPs are implemented as *quads*, with operands and results represented as temporary names (TNs). OPs can be queried through properties to allow code transformations such as code reselection through peephole optimizations.

structure member	description	
`srcpos`	source line number	general information
`opr`	target operation (TOP)	
`variant`	more abstract information for branch, compare, memory, and operations, etc	
`map_idx`	unique ID for an operation in a BB	
`scycle`	scheduling date	
`flags`	`OP_copy, OP_spill,` etc	
`next, prev`	sequential order	BB information
`bb`	BB where this op lives	
`unroll_bb`	unrolling: original BB and unrolled replication	
`order`	relative order in BB	
`results, opnds`	number of results and operands	arguments information
`res_opnd`	array of operands followed by results	

Table 8.8: Structure of OP in st200cc.

Temporary name. The compiler's semantic analysis phase will select registers for parameters and local variables, and choose machine-code addresses for procedure bodies. But it is too early to determine exactly which registers are available, or exactly where a procedure body will be located. We use TN as a temporary register to hold such values or as a tag to note constant values to hold information of addresses before they are really decided.

When TN denotes a register (`TN_is_register()`):

structure (union) member	description
`number`	register id
`save_creg`	callee saved register that this TN saves
`class_reg`	dedicated or allocated register ID and register class
`spill`	spill information if this TN is spilled
`home`	re-materialization information if this TN is re-materialize

Table 8.9: Register TN.

When TN holds constant values (`TN_is_constant()`):

values	inquiry function	structure (union) member
literal value	`TN_has_value()`	`value`: 64 bit literal value
enumeration	`TN_is_enum()`	`ecv`: enumeration description
label	`TN_is_label()`	`label`: label description `offset`: offset to label
symbol	`TN_is_symbol()`	`var`: symbol description `relocs`: relocations `offset`: offset to symbol

Table 8.10: Constant values of TN.

Example on building an operator. The listing below gives the expander function which creates CGIR operator for the equality operator (`==`). Here the operator is created at line 1 by function `Build_OP`, where `TOP_cmpeq_r_r_b` is the target operation, `b0`, `in0`, `in1`, and `ops` are temporary names.

Listing 8.14: Example of building the CGIR operator for equality operator.

```
0 void expand_eq(TN* in0, TN* in1, OPS* ops){
1   TN *b0 = Build_RCLASS_TN (ISA_REGISTER_CLASS_branch) ;
2   Build_OP (TOP_cmpeq_r_r_b,    b0,     in0,    in1,    ops) ;
3 }
```

8.3.2 Improvement of integer support for 64-bit

The ST231 compiler supports the C11 standard 'long long' type and its unsigned variant as a 64-bit integral type, emulated on the 32-bit architecture. As we have explained in § 3.4.1, when the emulation code is in the form of direct assembly emission ('open code'), the emission is done at reasonably high CGIR level and therefore it benefits from all further optimizations.

In this section we show an optimization made on the code emission of 64-bit unsigned min operator at CGIR, which leads to better code generation when one of the operand is some special constant.

The C expression of the 64-bit unsigned min operator is shown at line 1 of the listing below.

Listing 8.15: C function `minul` for 64-bit unsigned min.

```
0 uint64_t minul(uint64_t x, uint64_t y){
1   return (x < y)? x: y;
2 }
```

The CGIR operation expander function for minul is shown below:

Listing 8.16: Expanded CGIR operators for minul.

```
static void
Expand__minul( TN* ol0, TN* oh0, TN* il0, TN* ih0, TN* il1, TN* ih1, OPS* ops)
{
  TN *b0_0_0 = Build_RCLASS_TN (ISA_REGISTER_CLASS_branch);
  TN *b0_1_0 = Build_RCLASS_TN (ISA_REGISTER_CLASS_branch);
  TN *r0_20_0 = Build_RCLASS_TN (ISA_REGISTER_CLASS_integer);
  TN *r0_21_1 = Build_RCLASS_TN (ISA_REGISTER_CLASS_integer);
  Build_OP (TOP_cmpltu_r_r_b,    b0_0_0,       ih0,    ih1,    ops) ;
  Build_OP (TOP_cmpeq_r_r_b,     b0_1_0,       ih0,    ih1,    ops) ;
  Build_OP (TOP_minu_r_r_r,      r0_20_0,      il0,    il1,    ops) ;
  Build_OP (TOP_targ_slct_r_r_b_r,oh0,  b0_0_0, ih0,    ih1,    ops) ;
  Build_OP (TOP_targ_slct_r_r_b_r,r0_21_1,b0_0_0,       il0,    il1,    ops) ;
  Build_OP (TOP_targ_slct_r_r_b_r,ol0,  b0_1_0, r0_20_0,r0_21_1, ops) ;
}
```

Assembly code for Listing 8.15:

Listing 8.17: Assembly code for minl

```
cmpltu  $b1=$r17, $r19          ## (cycle 0)
cmpeq   $b0=$r17, $r19          ## (cycle 0)
minu    $r8=$r16, $r18          ## (cycle 0)
minu    $r17=$r17, $r19         ## (cycle 0)
 ;; ## (bundle 0)
slct    $r16=$b1, $r16, $r18    ## (cycle 1)
 ;; ## (bundle 1)
slct    $r16=$b0, $r8, $r16     ## (cycle 2)
return  $r63                    ## (cycle 2)
 ;; ## (bundle 2)
```

In our emulation for binary64 floating-point arithmetic, `minl` sometimes involves some special constants. For example, mul2 of binary64 format.

```
uint64_t absX = X & 0x7fffffffffffffffLL;
uint64_t Rgen = X + minul(absX, (uint64_t)1 << 0x34);
```

The lower 32-bit of the constant, `(uint64_t)1 << 0x34`, is zero. According to the CGIR expansion shown in Listing 8.16, the `il1` in Line 9 is zero. Then, we can conclude that `r0_20_0` is zero, which leads to the `minu` operation in Line 2 of Listing 8.17 to be removed. However, no more result can be propagated and the latency to evaluate the result remains the same.

Here, we propose a new expansion, which is shown below in Listing 8.18. Since `il1` in Line 2 of Listing 8.18 is zero, the result `r0_20_0` is zero. Then this result propagates to line 4, such that the result of selection `r0_21_1` is zero, which leads to a faster implementation shown in Listing 8.19.

Listing 8.18: Improved CGIR expansion for minl.

```
0 Build_OP (TOP_cmpeq_r_r_b,    b0_1_0,    ih0,    ih1,    ops);
1 Build_OP (TOP_cmpgeu_r_r_b,   b0_0_0,    ih0,    ih1,    ops);
2 Build_OP (TOP_minu_r_r_r,    r0_20_0,    il0,    il1,    ops);
3 Build_OP (TOP_targ_slct_r_r_b_r,  oh0,    b0_0_0,    ih1,    ih0,   ops);
4 Build_OP (TOP_targ_slct_r_r_b_r,  r0_21_1,b0_1_0,    r0_20_0, il1,   ops);
5 Build_OP (TOP_targ_slct_r_r_b_r,  ol0,    b0_0_0,    r0_21_1, il0,   ops);
```

Listing 8.19: Assembly generated by improved CGIR expansion.

```
cmpgeq  $b1=$r17, $r19          ## (cycle 0)
minu    $r17=$r17, $r19         ## (cycle 0)
 ;; ## (bundle 0)
slct    $r16=$b1, $r0, $r16     ## (cycle 1)
 ;; ## (bundle 1)
```

8.3.3 Integer range analysis framework

Overview of range analysis operating on SSA form The standard range analysis described in the literature (e.g. [Pat95b]) is an extension of the analysis performed for constant propagation. A popular algorithm is the Sparse Conditional Constant (SCC) algorithm [WZ91]; this algorithm operates on SSA form, and is used as the basis of the implementation described here.

The SSA algorithm is described in terms of a lattice, where a lattice value represents the range of possible values of a variable. There are special lattice values `TOP` and `BOTTOM`. `TOP` indicates an unvisited or uninitialized variable and `BOTTOM` indicates a variable with unknown value.

The basic algorithm for forward analysis to calculate the value range for all variables is:

- initialize all variables to `TOP`;
- add all instructions to a worklist;
- take an instruction from the worklist, and calculate the result range using the current range of the operands;

- add all instructions that use the result to the worklist, if the result range changes;
- continue until the worklist is empty.

A whole family of range analyses can be defined by making changes to the values held in the lattice:

- value range analysis, in which the lattice elements are `[min:max]` pairs of integers;
- an extension to value range analysis, in which the lattice elements are sets of `[min:max]` pairs;
- bit range analysis, in which the lattice elements are sign and number of significant bits;
- an analysis that may be useful for detecting alignments, in which the lattice value is the number of least-significant zeros. For example, a pointer value is 4-byte aligned if it has at least two least-significant zeros.

In all these cases, the framework is identical, it is just the lattice implementation that is changed.

It is also useful to have a backward analysis: this uses the same lattice, but visits nodes backward. This calculates the range of values that are required by the uses of a variable.

Implementation of integer range analysis framework on st200cc. This optimization is implemented at CGIR, because at this level both properties of the program and properties of the target are known, which enables the best possible precision for range analysis computations.

The framework consists of two phases: range analysis and range propagation (peephole optimizations). In the Open64 compiler, each phase is split into generic and target specific part.

Range analysis. First, a target specific part is called to handle target specific instructions. For instance the ST200 `clz` instruction creates values in the range of [0,32]. Then, a target independent part acts on generic instruction types based on standard Open64 compiler predicates.

Range propagation. After the range analysis has assigned a value range to each variable, this information is used by the range propagation phase to perform various code improvements, that are first target specific, and then generic.

Range propagation is indeed very similar to a “peepholing” transformation, where the knowledge of ranges on operands of operations enables more powerful and precise transformations.

8.3.4 Integer range analysis for shift operators

Here we detail a transformation made at the range propagation stage to expose better ILP. This work has been published in [BJJL+10, §6].

Motivation. For floating-point support on the ST231, we often compute L, specified in Equation (4.13), the mantissa of the result, by a right shift based on the sum of a format-related constant and a variable with predictable range. For example, L is computed as follows in FSA for binary32 format.

```
0 //uint32_t S,nlz,L;
1 //int32_t u;
2 nlz = countLeadingZeros(S);
3 u = max(3-nlz,0);
4 L = S >> (C + u); //C is a constant.
```

The right shift expression in line 4, $\texttt{S} \gg \texttt{(C + u)}$ (or similarly with a left shift), can be transformed into $\texttt{(S} \gg \texttt{C)} \gg \texttt{u}$, which improves the parallelism by relaxing the data dependency on `u`, provided that the following conditions hold:

$$u \in [0,31], \quad C \in [0,31], \quad u + C \in [0,31].$$

Then, instead of $\texttt{S} \gg \texttt{(C + u)}$ that incurs the following computations:

```
[1] computation of u
[2] tmp = C + u
[3] S >> tmp
```

we get a potentially better use of ILP (|| here means "in parallel with"):

```
[1] tmp = S >> C || computation of u
[2] tmp >> u
```

Implementation. Thanks to the range analysis framework, the implementation is very simple. At the propagation phase, when we first look for the following CGIR operation sequence.

```
    r2 = add r3  r4 (or immediate)
    r0 = shl (resp. shr(u)) r1  r2
```

Then if we have the information that **r2, r3**, and **r4** are in the range $[0,31]$ from the analysis phase, we replace the operations by

```
    r2 = shl (resp. shr(u)) r1  r4 (or immediate)
    r0 = shl (resp. shr(u)) r2  r3
```

In fact, this compilation transformation helped us to find similar situations during designing other floating-point operators and it is found to be applied in other benchmarks during validation.

8.3.5 Diverting integer range analysis for floating-point specialization

To select fused square-add (FSA) and addition of non-negative terms (addnn), we need specific optimization to prove the positivity of some operands during the compilation. Since an integer range analysis framework is already implemented in the compiler, we realize this optimization by diverting it for floating-point variables. Since the method is same for both FSA and addnn, we detail only the design of selecting FSA here.

Basic idea of selecting FSA. Listing 8.20 gives a typical routine to compute sum of squares (2-norm). To select the operation at line 4 as FSA, a specialized case of fused multiply-add (FMA) requires work in three steps.

1. Lowering FMA with squaring component to *general FSA* at WHIRL level;
2. Analyzing ranges of related floating-point variables by using the integer range analysis framework;
3. Selecting FSA from general FSA at integer range propagation phase.

The first step is done at lowering floating-point opcode (H WHIRL) to intrinsic opcode (M WHIRL), which is very similar to the selection of squaring and scaling. Steps 2 and 3 are done at CGIR level.

Listing 8.20: Sample code for FSA.

```
0 float sum(float a[]){
1   float s = 0.0f;
2   int i;
3   for(i=0;i<N;i++)
4     s += a[i]*a[i];
5   return s;}
```

Selecting general FSA. The general FSA refers to the pattern that $x \cdot x + z$ with no range requirement on z. To select this pattern, we check whether the two operands of the multiplication of a floating-point FMA are identical or not.

Same as squaring and scaling by a constant, call `WN_To_Intrinsic` to compute the intrinsic ID to lower High WHIRL (opcode = `OPC_F4MADD`) to Mid WHIRL (opcode = `OPC_F4INTRINSIC_OP`). The identity of the operands of the multiplication of an FMA is checked.

```
if(opcode == OPC_F4MADD){
  if (WN_Compare_Trees(kids[1],kids[2]) == 0) id = INTRN_SADDGS;
  else id = INTRN_MADDS;}
```

Then, we create the M WHIRL node. When the intrinsic code of the checked WHIRL tree is not that of an FMA, it is a general FSA. Then we need only two operands instead of three.

```
if((intrinsic_general!=intrinsic) && (type==MTYPE_F4)){
  nd = Extension_Aware_CreateIntrinsic(OPCODE_make_op(OPR_INTRINSIC_OP, res,
      MTYPE_V),
                        intrinsic, 2, kids);}
```

Analyzing the range of general FSA and selecting FSA. At CGIR level, we analyze the range of `s` of Listing 8.20 during the phase of range analysis and select the operation as an FSA when `s` is nonnegative. For a floating-point number, the MSB of its interchange encoding is zero when the value is nonnegative. As the range analysis is based on integer type, our work is to check if we can prove that the MSB of variable `s` is zero.

After SSA, line 4 of Listing 8.20 is eventually interpreted as

```
s’ = s+ a[i]*a[i];
s = s’;
```

Thus, to analyze the range of `s`, we must analyze the range of each general FSA. The listing below gives a general view of CGIR operators corresponding to the C code of Listing 8.20.

Listing 8.21: Excerpt CGIR operators corresponding to Listing 8.20

```
0 TOP_move r1 =  Const;
1 #loop head
2 ...
3 #loop body
4 TOP_call_i r = r1,r2
5 TOP_mov r1 = r
6 #loop tail
7 ...
```

Here, register r at line 4 holds the floating-point result of general FSA.

From the arithmetic point of view, floating-point numbers r_1 and r_2 satisfy,

$$r_1 + r_2 \cdot r_2 \geqslant r_1. \tag{8.1a}$$

Then, if we know the range of $r1 \in [\min, \max]$, we have

$$\text{range of}(r_1 + r_2 \cdot r_2) = [\min, \infty). \tag{8.1b}$$

To cast the floating-point range on the integer range analysis framework, we use the fact that

- when the result is a positive finite number or a positive infinity, the sign bit of the encoding is zero, which is required by the standard;
- when the input is a NaN, the result is a qNaN with MSB not changed, which is guaranteed by our implementation, although the sign bit of NaN results of general-computational operations are not specified by the IEEE 754 standard [IEE08, §6.3].

Thus, the range of $r1$ is denoted by a pair of unsigned integers $[\min, \max]$. We will apply the following rules to bound the range of each general FSA.

$$r_1 + r_2 \cdot r_2 \in [r_1, 0x7fffffff], \text{ when } r_1 \in [0, 0x7fffffff]. \tag{8.2}$$

The detailed C implementation for (8.2) is shown in the listing below.

Listing 8.22: Implementation of bounding the range of each general FSA.

```
0  if(opcode == TOP_call_i){
1    TN *opnd_op = OP_opnd(op, 0);
2    TN *opnd_r1 = OP_opnd(op, 1);
3    if(TN_is_symbol(opnd_op)){
4      if(strcmp(ST_name(TN_var(opnd_op)),"__saddgs")==0){
5        LRange_pc rref = lattice->makeRangeMinMax (0,0x7fffffff);
6        LRange_pc val1 = Value(opnd_r1);
7        if(rref->ContainsOrEqual(val1)){ //whether r1 is a positive number
8          if(val1->hasValue()) {
9            // r1 is a const, const + r2*r2 >= [const , inf)
10           new_value = lattice->makeRangeMinMax(val1->getValue(),0x7fffffff);
11         }else{
12           // range of (r1+r2*r2) = range of (r1)
13           new_value = Value(opnd_r1);
14         }
15         return TRUE;
16       }
17     }
18   }
19 }
```

Here, we first check at line 4 whether it is a general FSA operator. Then, we read the range of r_1 and compare if it is in the range of $[0, 0x7fffffff]$ at lines 6 and 7. If it is, we create the new range according to (8.2) to be linked with this general FSA operator from line 9 to 13.

Then in the range propagation stage, we check for each general FSA whether r_1 is in the range of $[0, 0x7fffffff]$ and replace the general by FSA if r_1 is in the good range. The C code is given in Appendix B.

8.4 Experimental results: UTDSP benchmark's FFT test suite

This section gives the details on the code generation of the production compiler for floating-point operations for the FFT test suite of the UTDSP benchmark [Lee]. This test suite provides complex radix-2 decimation-in-time 256-point and 1024-point FFT implementations, whose butterfly computation is floating-point intensive.

In our experiment the production compiler manages to select the most efficient custom floating-point operators for the butterflies, which leads to a speedup of 1.59x over the usage of general floating-point operators only.

Typical pattern of a radix-2 FFT butterfly. Listing 8.23 displays the C code of a radix-2 FFT butterfly in the test suite, whose pattern is typical for butterfly calculations from the view of compilation.

Listing 8.23: A radix-2 FFT butterfly in the UTDSP benchmark.

```
0  temp_real = Wr * data_real[2*j*buttersPerGroup+buttersPerGroup+k] -
1              Wi * data_imag[2*j*buttersPerGroup+buttersPerGroup+k];
2  temp_imag = Wi * data_real[2*j*buttersPerGroup+buttersPerGroup+k] +
3              Wr * data_imag[2*j*buttersPerGroup+buttersPerGroup+k];
4
5  sub0 = data_real[2*j*buttersPerGroup+k] - temp_real;
6  add0 = data_real[2*j*buttersPerGroup+k] + temp_real;
7  data_real[2*j*buttersPerGroup+buttersPerGroup+k] = sub0;
8  data_real[2*j*buttersPerGroup+k] = add0;
9
10 sub1 = data_imag[2*j*buttersPerGroup+k] - temp_imag;
11 add1 = data_imag[2*j*buttersPerGroup+k] + temp_imag;
12 data_imag[2*j*buttersPerGroup+buttersPerGroup+k] = sub1;
13 data_imag[2*j*buttersPerGroup+k] = add1;
```

Code selection by using general floating-point operators. Normally, `temp_real` in Line 0 of Listing 8.23 can be computed by two floating-point multiplication and one floating-point subtraction, or one multiplication and one FMA, shown in Listing 8.24. Here in line 2 of Listing 8.24, `__msubs` means fused multiply-subtract, $\circ(x \cdot y - z)$, which is essentially FMA. Respectively, `temp_imag` in line 2 can be computed by two multiplication and one addition, or one multiplication and one FMA (`__madds` at Line 2 of Listing 8.25).

Listing 8.24: Typical assembly code computing `temp_real` of the ST231.

```
0        call    $r63=__muls              ##   __muls
1         ;;
2        call    $r63=__msubs             ##   __msubs
3         ;;
```

Listing 8.25: Typical assembly code computing `temp_imag` of the ST231.

```
0        call    $r63=__muls              ##   __muls
1         ;;
2        call    $r63=__madds             ##   __madds
3         ;;
```

Meanwhile, `add0` and `sub0` (resp. `add1` and `sub1`) in Listing 8.23 are typically computed by one floating-point addition and one floating-point subtraction, shown in Listing 8.26.

Listing 8.26: Typically assembly code computing `sub0` and `add0` of the ST231.

```
        call    $r63=__adds              ##   __adds
         ;;
        call    $r63=__subs              ##   __subs
```

```
          ;;
```

Code selection by using custom floating-point operators. With the new production compiler, the computation of `temp_real` (resp. `temp_imag`) can be selected as `__dpsubs` (resp. `__dps`). The assembly code generated is given in Listing 8.27.

Listing 8.27: Assembly code computing `temp_real` and `temp_imag` by DP2.

```
call    $r63=__dpsubs            ##   __dpsubs
 ;;
call    $r63=__dps               ##   __dps
 ;;
```

The two calls in Listing 8.26 are merged to a single one to `__adspairs`, computing addsub pair, shown in Listing 8.28.

Listing 8.28: Assembly code computing `sub0` and `add0` by addsub.

```
call    $r63=__adspairs          ##   __adspairs
```

In this experiment, we see that a radix-2 FFT butterfly can be computed by using only custom floating-point operators, that is, each butterfly can be computed by 2 DP2 and 2 addsub.

Table 1.6 in Chapter 1 (reproduced below) gives the usage of custom operators in some test suites of the UTDSP benchmark and we observe that the production compiler achieves to select most of the custom operators from the source code.

	custom operators selected
FFT-256,1024	DP2 (50%), addsub (50%)
Latnrm-8	DP2 (67%), FMA (29%)
Latnrm-32	DP2 (66%), FMA (34%)
SPE	DP2 (20%), FMA (13%), addsub (9%), SOS (5%), square (1%)
ADPCM	FMA (25%), DP2 (8%), square (4%)
LPC	FMA (72%), DP2 (4%), square (2%), addsub (<1%)

Table 8.11: Custom operators selected.

Chapter 9

Conclusions and perspectives

In this thesis we have proposed a set of custom floating-point operators, and shown the benefits of having optimized software custom floating-point support to significantly speed up both individual calls and embedded applications on VLIW integer processors like the ST231. We have detailed the algorithms and implementations of squaring, scaling and its specializations, two-dimensional dot product and its specialization fused multiply-add, and simultaneous sine and cosine; we have also detailed the techniques developed in the compiler to select such operations in application codes.

This work has contributed to two complementary domains: *computer arithmetic designs* and *compiler optimizations*. For each of these domains, we present below our conclusions as well as some possible directions for future research.

Computer arithmetic designs

Over all the custom operators, we observe on the ST231 that for the binary32 format and $\circ = \mathrm{RN}$, the specialized operators introduce speedups ranging from 1.4 to 4.2 compared with their general operators of FLIP 1.0; the fused and paired operators provide speedups range from 1.02 to 1.95 compared with their naive implementations using general operators of FLIP 1.0. For all these operators, except FMA, we also observe significant reductions in code size. Similar improvements can be observed for other rounding modes as well.

Although our implementations are optimized for the binary32 format, the underlying design of all the operators has been parametrized by the format and generically described in an XML-based scheme. As a result, the implementations can be scaled to other formats like binary64 or binary128 as soon as the 64-bit or 128-bit integer arithmetic is supported.

On the ST231, the 64-bit integer layer exists naturally in the compiler which supports the ANSI C11 'long long' type, and for which lots of efforts have been done for code generation and optimization. Thanks to this support and to our parametrized approach, we obtain the C codes for the binary64 format without extra development cost - except for sincos. Table 9.1 gives the corresponding performances and, when comparing with the performances for the binary32 format (Table 1.1), we see that the latencies increase by a factor of 2 to 3.9 (DP2, RZ), which seems to be reasonable, since we expect a slowdown of 2 to 4 because of the cost of the emulation of 64-bit integer support compared to 32-bit native instructions.

On the other hand, although these codes for the binary64 format may be suboptimal, in our experiments we observe that for DP2 and FMA, they are at least 2x faster than the naive implementations using operators from the original ST library for double precision; for other custom operators the speedups are even more significant.

	RN		RU		RD		RZ	
mul2	9	[28]	11	[35]	12	[36]	11	[33]
div2	11	[22]	11	[34]	11	[34]	10	[29]
scaleB	28	[101]	29	[103]	29	[103]	22	[74]
square	26	[83]	25	[78]	22	[67]	22	[67]
andnn	29	[104]	30	[99]	26	[84]	26	[84]
FSA	58	[222]	56	[212]	45	[162]	45	[162]
FMA	115	[443]	111	[433]	110	[428]	104	[403]
SOS	65	[236]	63	[225]	53	[177]	53	[177]
DP2	182	[704]	178	[691]	178	[694]	170	[661]
addsub	65	[226]	70	[235]	70	[235]	64	[224]

Table 9.1: Performances for the binary64 format on the ST231 in # cycles [# instructions].

The work in computer arithmetic designs shows a number of directions that should be followed in the future:

A first direction deals with the sincos operator. Since our current design for sincos is for the binary32 format, the next step would be to extend it to the binary64 format. This can be achieved by using the same software toolchain to find a new polynomial approximation and to generate its certified evaluation scheme. On the other hand, we would also like to provide a parametrized design for range reduction: so far we have assumed an input in $[-\pi/4, \pi, 4]$, but for large inputs it remains to be studied how reduction to that range can be performed accurately enough and with high ILP exposure for the binary$k$ format on VLIW integer architectures.

As a second direction, we should also investigate algorithm designs and their optimized implementations for more floating-point operators. For example, correctly-rounded sums of more than two terms, like $x + y + z$ or $x + y + z + t$, could be worth being implemented. Also, one step beyond sum of squares (SOS) would be to compose it with square root to get a correctly-rounded hypotenuse function $\sqrt{x^2 + y^2}$, as recommended by the standard. Finally, its three-dimensional counterpart $\sqrt{x^2 + y^2 + z^2}$ should be interesting for 3D-graphics applications. In all these cases, although high speedups can hardly be expected, the latencies should still be lower than the ones of the naive implementations and, more importantly, the impact on the accuracy, especially for sums of many terms, could be significant.

A third direction would be to study further the impact on performances (latency, code size) of relaxing the IEEE 754 specification used so far. By relaxed specifications, we mean variants such as *finite-math-only* (which discards ∞, NaN, and overflow) or *without subnormals*, as well as variants having a proven accuracy of only a few ulps. In the special case of squaring, we have already seen that at most one cycle is saved for relaxed variants like finite-math-only or without subnormals. We can expect similar conclusions for our other custom operators, but this remains to be done. Also—and this seems more challenging, what exactly can we gain when relaxing the accuracy constraints? For example, OpenCL [Ope] requires only 4 ulps of accuracy for sine and cosine, but it is not clear if a significant speedup (by a factor of 2, say) is then possible. Ideally, to explore this, we should have designs parametrized not only by the format, as we do now, but also by the targeted accuracy.

Compiler optimizations

The optimizations to select custom floating-point operators and to ensure the best possible code selection when compiling the implementations of each floating-point operator described in this thesis have been implemented in a specific variant of the Open64 compiler for the ST231 target. Along with the libgcc library including the implementation of the specialized operators, this project passes successfully the whole industrial test suite mandatory to deliver this improved compiler to end-users. To our knowledge, this is the first time that a floating-point range analysis technique is implemented and used for optimization in an industrial, production-quality compiler.

Thanks to this production compiler, real applications can benefit from the custom floating-point operators proposed in this thesis. We have seen that the UTDSP benchmark's FFT test suites can be fully performed by DP2 and addsub operations without using other general operators. The complete support of custom operators introduces speedups ranging from 1.17 to 1.59 on various UTDSP kernels and applications (see Table 1.5). For the graphics applications, we have also observed speedups from 1.13 to 1.45 by comparison with using only general operators provided by FLIP 1.0 (see Table 6.6).

The work in compiler optimizations shows a number of directions that should be followed in the future:

A first direction is to deal with the weaknesses in the current optimizations. The selection of custom operators is not exempt from a few open issues, which are fortunately not exhibiting functional issues: for instance we have observed, but not been able to fix so far, that the selection of arithmetic operators pairs was very sensitive to the shapes of the expression used and thus sometimes missed. Another issue is that, from the user standpoint, the control of fused floating-point operation selection through global options is not precise enough in its scope. Ideally, our implementation should account for the C11 pragma `FP_CONTRACT` semantic, as described in the ISO/IEC 9899:2011 standard [Int11]: the contraction of expressions would be naturally allowed or disallowed in the regions controlled by this pragma.

The second direction would be to implement the native 128-bit integer support in the compiler. Since our design for floating-point operators is fully parametrized, on the ST231, the support of the binary128 format reduces to the support of 128-bit integer arithmetic. The 128-bit support exists only as a prototype library with no native implementation in the compiler: thus at the moment it is not as efficient as it could be, suffering from the representation of 128-bit types by structures. Ideally, implementing the native support for the `__uint128_t` and `__int128_t` gcc extensions (at the moment limited to 64-bit machines) would alleviate these issues.

As a third direction, we would like to improve the range analysis techniques for floating-point operations. First, the current simple technique based on integer range analysis could be further extended to catch more positivity cases. For this, an immediately usable idea would be to augment the precision of the analysis by recognizing elementary functions calls in the libm and propagating range information from them. Furthermore, the implementation level choices done in this work (WHIRL vs CGIR) could be revisited, in particular, a range analysis framework could be implemented at the WHIRL level, thus benefiting to all the compiler targets, but it remains to be proved if the information available would be sufficient to compute information equivalent to what we get at the CGIR level. Finally, we think that the next key enabler to implement more compilation-time floating-point optimizations is the availability of a pure floating point range analysis framework in the compiler, possibly helped by user-level annotations, specifically designed to use various numerical abstract domains, and defining properties of all the libm functions. This would allow the exploration of new floating-point optimizations, such as the specialization of libm entry points to specific domains at their point of use.

Bibliography

[AC00] Tor Aamodt and Paul Chow. Embedded ISA support for enhanced floating-point to fixed-point ANSI-C compilation. In *Proceedings of the 2000 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES'00, pages 128–137, New York, NY, USA, 2000. ACM. [cited on page(s) 1]

[ADN07] Christian Artigues, Sophie Demassey, and Emmanuel Neron. *Resource-Constrained Project Scheduling: Models, Algorithms, Extensions and Applications*. ISTE, 2007. [cited on page(s) 27]

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006. [cited on page(s) 139]

[BDR$^+$09] Benoit Boissinot, Alain Darte, Fabrice Rastello, Benoit Dupont de Dinechin, and Christophe Guillon. Revisiting out-of-SSA translation for correctness, code quality and efficiency. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO'09, pages 114–125, Washington, DC, USA, 2009. IEEE Computer Society. [cited on page(s) 27]

[Bet08] Timo Betcke. Optimal scaling of generalized and polynomial eigenvalue problems. *SIAM J. Matrix Anal. Appl.*, 30(4):1320–1338, 2008. [cited on page(s) 60]

[BGS00] Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. Abcd: eliminating array bounds checks on demand. In *PLDI*, pages 321–333, 2000. [cited on page(s) 3]

[BJJL$^+$10] Christian Bertin, Claude-Pierre Jeannerod, Jingyan Jourdan-Lu, Hervé Knochel, Christophe Monat, Christophe Mouilleron, Jean-Michel Muller, and Guillaume Revy. Techniques and tools for implementing IEEE 754 floating-point arithmetic on VLIW integer processors. In *Proceedings of PASCO'10*, pages 1–9. ACM, 2010. [cited on page(s) 2, 5, 31, 144]

[Blu78] James L. Blue. A portable Fortran program to find the Euclidean norm of a vector. *ACM Trans. Math. Soft.*, 4(1):15–23, 1978. [cited on page(s) 54, 56]

[Bru09] Christian Bruel. If-conversion for embedded VLIW architectures. *International Journal of Embedded Systems (IJES)*, 4(1):2–16, 2009. [cited on page(s) 27, 29]

[CC93] W. J. Cody and Jerome T. Coonen. Algorithm 722: Functions to support the IEEE standard for binary floating-point arithmetic. *ACM Trans. Math. Soft.*, 19(4):443–451, 1993. [cited on page(s) 61, 80]

[CC99] Andrea G. M. Cilio and Henk Corporaal. Floating point to fixed point conversion of C code. In *Proceedings of the 8th International Conference on Compiler Construction, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99*, CC'99, pages 229–243, London, UK, UK, 1999. Springer-Verlag. [cited on page(s) 1]

[CHT02] Marius Cornea, John Harrison, and Ping Tak Peter Tang. *Scientific Computing on Itanium®-based Systems.* Intel Press, 2002. [cited on page(s) 2, 113]

[CJL10] Sylvain Chevillard, Mioara Joldes, and Christoph Lauter. Sollya: an environment for the development of numerical codes. In *Proc. of the Third International Congress on Mathematical Software (ICMS)*, pages 28–31. LNCS, Springer, 2010. [cited on page(s) 114, 117]

[CP07] Yee Jern Chong and Sri Parameswaran. Automatic application specific floating-point unit generation. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '07, pages 461–466, San Jose, CA, USA, 2007. EDA Consortium. [cited on page(s) 2]

[CP09] Yee Jern Chong and Sri Parameswaran. Custom floating-point unit generation for embedded systems. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 28(5):638–650, May 2009. [cited on page(s) 2]

[crl] CR-Libm, a library of correctly rounded elementary functions in double precision. Available at `http://lipforge.ens-lyon.fr/www/crlibm/`. [cited on page(s) 113]

[dD07] Benoit Dupont de Dinechin. Time-indexed formulations and a large neighborhood search for the resource-constrained modulo scheduling problem. In *3rd Multidisciplinary International Scheduling conference: Theory and Applications (MISTA)*, 2007. [cited on page(s) 27]

[DdD07] Jérémie Detrey and Florent de Dinechin. Floating-point trigonometric functions for FPGAs. In *Proc. IEEE International Conference on Field-Programmable Logic and Applications (FPL)*, pages 29–34, 2007. [cited on page(s) 113]

[dDP11] Florent de Dinechin and Bogdan Pasca. Designing custom arithmetic data paths with FloPoCo. *IEEE Design & Test of Computers*, 28(4):18–27, July 2011. [cited on page(s) 2]

[Dem84] J. Demmel. Underflow and the reliability of numerical software. *SIAM Journal on Scientific and Statistical Computing*, 5(4):887–919, 1984. [cited on page(s) 20]

[EL04] M. D. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kaufmann, 2004. [cited on page(s) 21, 42, 89, 90]

[FBF$^+$00] Paolo Faraboschi, Geoffrey Brown, Joseph A. Fisher, Giuseppe Desoli, and Fred Homewood. Lx: a technology platform for customizable VLIW embedded processing. In *Proc. of the 27th International Symposium on Computer Architecture (ISCA)*, pages 203–213. ACM, 2000. [cited on page(s) 23]

[fdl] FdLibM: C math library for machines that support IEEE 754 floating-point. Available at `http://www.netlib.org/fdlibm/`. [cited on page(s) 61]

[GB91] Shmuel Gal and Boris Bachelis. An accurate elementary mathematical library for the IEEE floating point standard. *ACM Trans. Math. Softw.*, 17:26–45, March 1991. [cited on page(s) 113]

[GKP94] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, Reading, MA, USA, second edition, 1994. [cited on page(s) 47]

[Gök08] Mustafa Gök. Integer squarers with overflow detection. *Computers & Electrical Engineering*, 34(5):378–391, 2008. [cited on page(s) 42]

[Gol91] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991. [cited on page(s) 13]

[GRBB05] Christophe Guillon, Fabrice Rastello, Thierry Bidault, and Florent Bouchez. Procedure placement using temporal-ordering information: Dealing with code size expansion. *J. Embedded Comput.*, 1(4):437–459, December 2005. [cited on page(s) 26]

[Hau] John Hauser. The SoftFloat and TestFloat Packages. Available at `http://www.jhauser.us/arithmetic/`. [cited on page(s) 1]

[Hau96] John R. Hauser. Handling floating-point exceptions in numeric programs. *ACM Transactions on Programming Languages and Systems*, 18:139–174, 1996. [cited on page(s) 20]

[HH05] Desmond J. Higham and Nicholas J. Higham. *MATLAB Guide Second Edition*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2005. [cited on page(s) 60, 81]

[Hig02] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadelphia, PA, USA, second edition, 2002. [cited on page(s) 4, 13, 41, 56, 86]

[HPW90] Eldon R. Hansen, Merrell L. Patrick, and Richard L. C. Wang. Polynomial evaluation with scaling. *ACM Trans. Math. Softw.*, 16(1):86–93, 1990. [cited on page(s) 59]

[IEE08] IEEE Computer Society. *IEEE Standard for Floating-Point Arithmetic*. IEEE Standard 754-2008, August 2008. [cited on page(s) 1, 2, 3, 4, 13, 14, 15, 16, 18, 19, 42, 44, 60, 86, 87, 88, 128, 147]

[Int99] International Organization for Standardization. *Programming Languages – C*. ISO/IEC Standard 9899:1999, Geneva, Switzerland, December 1999. [cited on page(s) 47]

[Int11] International Organization for Standardization. *Programming Languages – C*. ISO/IEC Standard 9899:201x, Geneva, Switzerland, April 2011. [cited on page(s) 60, 153]

[JJL12] Claude-Pierre Jeannerod and Jingyan Jourdan-Lu. Simultaneous floating-point sine and cosine for VLIW integer processors. In *Proceedings of the 23rd IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2012. [cited on page(s) 11, 113]

[JJLMR11] Claude-Pierre Jeannerod, Jingyan Jourdan-Lu, Christophe Monat, and Guillaume Revy. How to square floats accurately and efficiently on the ST231 integer processor. In 10.1109/ARITH.2011.19, editor, *Proceedings of the 20th IEEE Symposium on Computer Arithmetic (ARITH'20)*, pages 77–81, Tübingen, Germany, July 2011. IEEE Computer Society. [cited on page(s) 10, 41]

[JKMR11] Claude-Pierre Jeannerod, Hervé Knochel, Christophe Monat, and Guillaume Revy. Computing floating-point square roots via bivariate polynomial evaluation. *IEEE Trans. on Computers*, 60(2):214–227, 2011. [cited on page(s) 116, 121]

[JLM12] Claude-Pierre Jeannerod, Nicolas Louvet, and Jean-Michel Muller. Further analysis of Kahan's algorithm for the accurate computation of 2×2 determinants. *Mathematics of Computation*, 2012. to appear. Preliminary version available at `http://hal-ens-lyon.archives-ouvertes.fr/ensl-00649347/en/`. [cited on page(s) 86]

[JR09a] Claude-Pierre Jeannerod and Guillaume Revy. FLIP 1.0: a fast floating-point library for integer processors. `http://flip.gforge.inria.fr/`, February 2009. [cited on page(s) 1, 52]

[JR09b] 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, November 2009. [cited on page(s) 2]

[Kah81] W. Kahan. Why do we need a floating-point arithmetic standard?, 1981. [cited on page(s) 13]

[Kah96] W. Kahan. Lecture notes on the status of IEEE Standard 754 for binary floating-point arithmetic. Manuscript, May 1996. [cited on page(s) 4, 20]

[Kah98] W. Kahan. Matlab's loss is nobody's gain. Available at `http://www.cs.berkeley.edu/~wkahan/MxMulEps.pdf`, 1998. [cited on page(s) 86]

[Knu87] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, third edition, 1987. [cited on page(s) 118]

[Lee] Corinna G. Lee. UTDSP Benchmark Suite. Available at `http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html`. [cited on page(s) 8, 111, 148]

[LS07] Karlo Gusso Lenzi and Osamu Saotome. Optimized math functions for a fixed-point DSP architecture. In *19th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 125–132, 2007. [cited on page(s) 113]

[Mar00] Peter Markstein. *IA-64 and Elementary Functions: Speed and Precision*. Prentice-Hall, Englewood Cliffs, NJ, 2000. [cited on page(s) 113]

[Mar03] Peter Markstein. Accelerating sine and cosine evaluation with compiler assistance. In *Proc. of the 16th IEEE Symposium on Computer Arithmetic (ARITH)*, pages 137–140. IEEE Computer Society, 2003. [cited on page(s) 2, 113]

[MBdD$^+$10] 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, 2010. [cited on page(s) 13, 14, 19, 21, 50, 89, 90, 98]

[MCCS02] Daniel Ménard, Daniel Chillet, Franois Charot, and Olivier Sentieys. Automatic floating-point to fixed-point conversion for DSP code generation. In *Proceedings of the 2002 international conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 270–276, 2002. [cited on page(s) 1]

[MCS05] A. Mitra, M. Chakraborty, and H. Sakai. A block floating-point treatment to the LMS algorithm: efficient realization and a roundoff error analysis. *IEEE Transactions on Signal Processing*, 53(12):4536–4544, 2005. [cited on page(s) 1]

[Mel] Guillaume Melquiond. Gappa - génération automatique de preuves de propriétés arithmétiques. `http://gappa.gforge.inria.fr/`. [cited on page(s) 114, 117]

[MR11] Christophe Mouilleron and Guillaume Revy. Automatic generation of fast and certified code for polynomial evaluation. In *Proc. of the 20th IEEE Symposium on Computer Arithmetic (ARITH)*, pages 233–242. IEEE Computer Society, 2011. [cited on page(s) 114, 122]

[MRS$^+$01] Scott Mahlke, Rajiv Ravindran, Michael Schlansker, Robert Schreiber, and Timothy Sherwood. Bitwidth cognizant architecture synthesis of custom hardware accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20:1355–1371, 2001. [cited on page(s) 3]

[Mul05] Jean-Michel Muller. On the definition of ulp(x). Technical Report 2005-09, École normale supérieure de Lyon, Laboratoire de l'Informatique du Parallélisme, 2005. [cited on page(s) 21]

[Mul06] Jean-Michel Muller. *Elementary Functions, Algorithms and Implementation.* Birkhäuser Boston, MA, USA, second edition, 2006. [cited on page(s) 113]

[Ope] OpenCL 1.2 Specification. version 15, released November 15, 2011, http://www.khronos.org/registry/cl/. [cited on page(s) 153]

[Ope00a] WHIRL Intermediate Language Specification, October 2000. Available from http://www.open64.net/documentation/manuals.html. [cited on page(s) 128, 129]

[Ope00b] WHIRL Symbol Table Specification, October 2000. Available from http://www.open64.net/documentation/manuals.html. [cited on page(s) 130]

[Pat95a] Jason R. C. Patterson. Accurate static branch prediction by value range propagation. *SIGPLAN Not.*, 30(6):67–78, June 1995. [cited on page(s) 3]

[Pat95b] Jason R. C. Patterson. Accurate static branch prediction by value range propagation. *SIGPLAN Not.*, 30(6):67–78, June 1995. [cited on page(s) 143]

[PKS00] Vassilis Paliouras, Konstantina Karagianni, and Thanos Stouraitis. A floating-point processor for fast and accurate sine/cosine evaluation. *IEEE Trans. on Circuits and Systems - Part II: Analog and Digital Signal Processing*, 47(5):441–451, 2000. [cited on page(s) 113]

[PR69] B. N. Parlett and C. Reinsch. Balancing a matrix for calculation of eigenvalues and eigenvectors. *Numerische Mathematik*, 13(4):293–304, 1969. [cited on page(s) 60]

[Pri04] Douglas M. Priest. Efficient scaling for complex division. *ACM Trans. Math. Softw.*, 30(4):389–401, 2004. [cited on page(s) 59]

[PTVF07] William Press, Saul Teukolsky, William Vetterling, and Brian Flannery. *Numerical Recipes Third Edition: The Art of Scientific Computing*. Cambridge University Press, 32 Avenue of the Americas, NY 10013-2473, USA, 2007. [cited on page(s) 41, 59, 60, 81]

[Pyt] Python v2.7.3 documentation. [cited on page(s) 37]

[Rai06] Saurabh-Kumar Raina. *FLIP: a Floating-point Library for Integer Processors*. PhD thesis, ENS de Lyon, France, September 2006. [cited on page(s) 42, 113, 114]

[Rev09] Guillaume Revy. *Implementation of binary floating-point arithmetic on embedded integer processors: polynomial evaluation-based algorithms and certified code generation*. PhD thesis, Université de Lyon - ÉNS de Lyon, France, December 2009. [cited on page(s) 1, 2, 20, 54]

[Sal09] Hani H. Saleh. *Fused Floating-Point Arithmetic For DSP*. PhD thesis, The University of Texas at Austin, May 2009. [cited on page(s) 2]

[SEES08] Hani H. Saleh and Jr Earl E. Swartzlander. A floating-point fused dot-product unit. In *2008 IEEE International Conference on Computer Design (ICCD 2008)*, pages 427–431, Lake Tahoe, Canada, 2008. [cited on page(s) 2, 86]

[SGPB11] Rodrigo Sol, Christophe Guillon, Fernando Magno Quintão Pereira, and Mariza A. S. Bigonha. Dynamic elimination of overflow tests in a trace compiler. In *Proceedings of the 20th international conference on Compiler construction: part of the joint European conferences on theory and practice of software*, CC'11/ETAPS'11, pages 2–21, Berlin, Heidelberg, 2011. Springer-Verlag. [cited on page(s) 3]

[Shi10] Naoki Shibata. Efficient evaluation methods of elementary functions suitable for SIMD computation. *Computer Science - Research and Development*, 25(1-2):25–32, 2010. [cited on page(s) 113]

[Sim08] Axel Simon. *Value-Range Analysis of C Programs: Towards Proving the Absence of Buffer Overflow Vulnerabilities*. Springer Publishing Company, Incorporated, 1 edition, 2008. [cited on page(s) 3]

[SS12] Earl E. Swartzlander and Hani H. M. Saleh. FFT implementation with fused floating-point operations. *IEEE Trans. on Computers*, 61(2):284–288, 2012. [cited on page(s) 2, 86]

[SSDT05] Eric M. Schwarz, Martin Schmookler, and Son Dao Trong. FPU implementations with denormalized numbers. *IEEE Trans. Comput.*, 54(7):825–836, July 2005. [cited on page(s) 20]

[ST209] ST200 VLIW Series - ST200 run-time architecture manual, June 2009. [cited on page(s) 29]

[Ste74] P. H. Sterbenz. *Floating-point computation.* Prentice-Hall series in automatic computation. Prentice-Hall, 1974. [cited on page(s) 20]

[Tan90] Ping Tak Peter Tang. Some software implementations of the functions sine and cosine. Technical Report ANL-90/3, Argonne National Laboratory, Argonne, Ill., April 1990. [cited on page(s) 113]

[Tis06] Arnaud Tisserand. Hardware operator for simultaneous sine and cosine evaluation. In *Proc. IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, volume 3, pages 992–995, 2006. [cited on page(s) 113]

[VB11] Álvaro Vázquez and Javier D. Bruguera. Composite iterative algorithm and architecture for q-th root calculation. In *Proceedings of the 20th IEEE Symposium on Computer Arithmetic (ARITH-20)*, pages 52–61, Tübingen, Germany, July 2011. [cited on page(s) 2]

[W3C] W3C. Extensible Markup Language (XML) 1.0 (Fifth Edition). [cited on page(s) 37]

[WSS01] E. G. Walters, J. Schlessman, and M. J. Schulte. Combined unsigned and two's complement hybrid squarers. In *Proceedings of the thirty-fifth Conference on Signals, Systems, and Computers (Asilomar 2001)*, volume 1, pages 861–866, Asilomar, Pacific Grove, CA , USA, 2001. IEEE. [cited on page(s) 42]

[WZ91] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, April 1991. [cited on page(s) 143]

A Some C implementations for various integer functions

Implementation of 32-bit min operators.

```
inline int32_t min(int32_t x, int32_t y){
 return (x<y)?x:y;
}
```

```
inline uint32_t minu(uint32_t x, uint32_t y){
 return (x<y)?x:y;
}
```

Implementation of 32-bit max operators.

```
inline int32_t max(int32_t x, int32_t y){
 return (x>y)?x:y;
}
```

```
inline uint32_t maxu(uint32_t x, uint32_t y){
 return (x>y)?x:y;
}
```

Implementation of unsigned $32 \times 32 \rightarrow 32$ multiplication.

```
inline uint32_t mul(uint32_t x, uint32_t y){
  uint64_t a = x;
  uint64_t b = y;
  uint64_t r = (a * b) >> 32;
  return r;
}
```

Implementation of unsigned 64-bit multiplications, `mulh64`, `mull64`.

```
static inline uint32_t mulh32(uint32_t x,uint32_t y){
  uint64_t r =  (uint64_t)x*(uint64_t)y;
  return (uint32_t)(r>>32);
}

static inline uint32_t mull32(uint32_t x,uint32_t y){
  uint64_t r =  (uint64_t)x*(uint64_t)y;
  return (uint32_t)r;
}

uint64_t mulh64(uint64_t x, uint64_t y){
  uint32_t x1,x0,y1,y0,xy11h,xy11l,xy10h,xy10l,xy01h,xy01l,xy00h;
  uint64_t A,B,C,Rh,Rl;

  x0 = (uint32_t)x;
  x1 = (uint32_t)(x>>32);
  y0 = (uint32_t)y;
  y1 = (uint32_t)(y>>32);


  xy10l = mull32(x1,y0);
  xy01l = mull32(x0,y1);

  xy10h = mulh32(x1,y0);
  xy01h = mulh32(x0,y1);

  xy00h = mulh32(x0,y0);

  xy11l = mull32(x1,y1);
  xy11h = mulh32(x1,y1);

  A = __st200addcg(xy10l,xy01l,0);

  B = __st200addcg(xy10h,xy01h,A>>32);

  C = __st200addcg(xy00h,(uint32_t)A,0);

  Rl= __st200addcg(xy11l,(uint32_t)B,C>>32);

  Rh= __st200addcg(xy11h,(uint32_t)(B>>32),Rl>>32);

  return (Rh<<32)|((Rl<<32)>>32);

}
```

```
inline uint64_t mull64(uint64_t x, uint64_t y){
  return x*y;
}
```

Implementation of unsigned 128-bit multiplications, `mulh128`, `mull128`.

```
typedef struct {
  uint64_t l;
  uint64_t h;
}uint128_t;

static inline uint64_t mul32(uint32_t x, uint32_t y){
  uint64_t r = (uint64_t)x*(uint64_t)y;
  return r;
}

inline uint128_t mulh128(uint128_t u, uint128_t v){
  /*!
    (a*2^(96)+b*2^(64)+c*2^(32)+d) *(e*2^(96)+f*2^(64)+g*2^(32)+h)
   = (ae)*2^(192) +(af+be)*2^(160)+(ag+ce+bf)*2^(128)
    +(ah+de+bg+cf)*2^(96)+(bh+df+cg)*2^(64)+(ch+dg)*2^(32)+dh
   */

  uint32_t a,b,c,d,e,f,g,h;
  uint64_t r7,r6,r5,r4,r3,r2,r1,r0;
  uint64_t ae,af,ag,ah,be,bf,bg,bh,ce,cf,cg,ch,de,df,dg,dh;
  uint128_t r;
  a = (uint32_t)(u.h>>32);
  b = (uint32_t)((u.h<<32)>>32);
  c = (uint32_t)(u.l>>32);
  d = (uint32_t)((u.l<<32)>>32);

  e =  (uint32_t)(v.h >>32);
  f =  (uint32_t)((v.h <<32)>>32);
  g =  (uint32_t)(v.l >>32);
  h =  (uint32_t)((v.l<<32)>>32);

  dh= mul32(d,h);
  ch= mul32(c,h);
  bh= mul32(b,h);
  ah= mul32(a,h);

  dg= mul32(d,g);
  cg= mul32(c,g);
  bg= mul32(b,g);
  ag= mul32(a,g);

  df= mul32(d,f);
  cf= mul32(c,f);
  bf= mul32(b,f);
  af= mul32(a,f);

  de= mul32(d,e);
  ce= mul32(c,e);
  be= mul32(b,e);
  ae= mul32(a,e);

 /*!
    (a*2^(96)+b*2^(64)+c*2^(32)+d) *(e*2^(96)+f*2^(64)+g*2^(32)+h)
  = (ae)*2^(192) +(af+be)*2^(160)+(ag+ce+bf)*2^(128)
   +(ah+de+bg+cf)*2^(96)+(bh+df+cg)*2^(64)+(ch+dg)*2^(32)+dh
  */

  r0 = (dh<<32)>>32;
  r1 = ((ch<<32)>>32)+((dg<<32)>>32)+(dh>>32);
  r2 =((bh<<32)>>32)+((df<<32)>>32)+((cg<<32)>>32)+(ch>>32)+(dg>>32)+(r1>>32);
  r3 =((ah<<32)>>32)+((de<<32)>>32)+((bg<<32)>>32)+((cf<<32)>>32)+(bh>>32)+(df
      >>32)+(cg>>32)+(r2>>32);
  r4 =((ag<<32)>>32)+((ce<<32)>>32)+((bf<<32)>>32)+(ah>>32)+(de>>32)+(bg>>32)
      +(cf>>32)+(r3>>32);
  r5 =((af<<32)>>32)+((be<<32)>>32)+(ah>>32)+(de>>32)+(bg>>32)+(cf>>32)+(r4
      >>32);
  r6 =((ae<<32)>>32)+(af>>32)+(be>>32)+(r5>>32);
  r7 =(ae>>32)+(r6>>32);

  r.h = (r7<<32)|((r6<<32)>>32);
  r.l = (r5<<32)|((r4<<32)>>32);

  return r;
}
```

```
inline uint128_t mull128(uint128_t u, uint128_t v){
  /*!
    (a*2^(96)+b*2^(64)+c*2^(32)+d) *(e*2^(96)+f*2^(64)+g*2^(32)+h)
  = (ae)*2^(192) +(af+be)*2^(160)+(ag+ce+bf)*2^(128)
   +(ah+de+bg+cf)*2^(96)+(bh+df+cg)*2^(64)+(ch+dg)*2^(32)+dh
  */

  uint32_t a,b,c,d,e,f,g,h;
  uint64_t r7,r6,r5,r4,r3,r2,r1,r0;
  uint64_t ae,af,ag,ah,be,bf,bg,bh,ce,cf,cg,ch,de,df,dg,dh;
  uint128_t r;
```

```
 a = (uint32_t)(u.h>>32);
 b = (uint32_t)((u.h<<32)>>32);
 c = (uint32_t)(u.l>>32);
 d = (uint32_t)((u.l<<32)>>32);

 e =  (uint32_t)(v.h >>32);
 f =  (uint32_t)((v.h <<32)>>32);
 g =  (uint32_t)(v.l >>32);
 h =  (uint32_t)((v.l<<32)>>32);

 dh= mul32(d,h);
 ch= mul32(c,h);
 bh= mul32(b,h);
 ah= mul32(a,h);

 dg= mul32(d,g);
 cg= mul32(c,g);
 bg= mul32(b,g);
 ag= mul32(a,g);

 df= mul32(d,f);
 cf= mul32(c,f);
 bf= mul32(b,f);
 af= mul32(a,f);

 de= mul32(d,e);
 ce= mul32(c,e);
 be= mul32(b,e);
 ae= mul32(a,e);

/*!
    (a*2^(96)+b*2^(64)+c*2^(32)+d)  *(e*2^(96)+f*2^(64)+g*2^(32)+h)
  = (ae)*2^(192)  +(af+be)*2^(160)+(ag+ce+bf)*2^(128)
    +(ah+de+bg+cf)*2^(96)+(bh+df+cg)*2^(64)+(ch+dg)*2^(32)+dh
  */

 r0 = (dh<<32)>>32;
 r1 = ((ch<<32)>>32)+((dg<<32)>>32)+(dh>>32);
 r2 =((bh<<32)>>32)+((df<<32)>>32)+((cg<<32)>>32)+(ch>>32)+(dg>>32)+(r1>>32);
 r3 =((ah<<32)>>32)+((de<<32)>>32)+((bg<<32)>>32)+((cf<<32)>>32)+(bh>>32)+(df
     >>32)+(cg>>32)+(r2>>32);
 r4 =((ag<<32)>>32)+((ce<<32)>>32)+((bf<<32)>>32)+(ah>>32)+(de>>32)+(bg>>32)
     +(cf>>32)+(r3>>32);
  r5 =((af<<32)>>32)+((be<<32)>>32)+(ah>>32)+(de>>32)+(bg>>32)+(cf>>32)+(r4
      >>32);
  r6 =((ae<<32)>>32)+(af>>32)+(be>>32)+(r5>>32);
  r7 =(ae>>32)+(r6>>32);

  r.h = (r3<<32)|((r2<<32)>>32);
  r.l = (r1<<32)|((r0<<32)>>32);

  return r;
}
```

B Some C codes for compiler optimizations

Selecting DP2 for n-dimensional dot products (§8.2.3, page 136).

```
// lowering a chain of FMA to DP2.
//
// eg: r = a*b +c*d + e*f + g*h
//      r = madd(madd( dp(a,b,c,d),e,f),g,h)
//==>> r = add(dp (a,b,c,d), dp(e,f,g,h))

static void lower_madd_tree(WN *block, WN *tree, LOWER_ACTIONS actions);
static WN *lower_one_madd_tree(WN *block, WN *wn);
static WN *sum_madd_segments(WN **segments, INT high, INT low);

// step1: search an FMA chain
static void lower_madd_tree(WN *block, WN *tree, LOWER_ACTIONS actions)
{
  if (WN_operator(tree) != OPR_MADD) {
    INT16 i;
    for (i = 0; i < WN_kid_count(tree); i++) {
      WN *kid = WN_kid(tree, i);
      if (WN_operator(kid) == OPR_MADD) {
        // Kid potentially begins a MADD chain.
        WN_kid(tree, i) = lower_one_madd_tree(block, kid);
      } else {
        lower_madd_tree(block, kid, actions);
      }
    }
  }
}

// step 2: split a single madd tree
// for the example expr, segments[0] = dp(e,f,g,h), segments[1]=dp(a,b,c,d)
static WN *lower_one_madd_tree(WN *block, WN *wn)
{
  // Tree is a MADD and potentially starts a MADD chain.
  Is_True(WN_operator(wn) == OPR_MADD,
          ("lower_one_madd_tree_height: MADD operator not found"));

  // Count the MADDs in the chain.
  INT madd_count = 0;
  WN *this_wn;
  TYPE_ID  type = WN_rtype(wn);
  for (this_wn = wn;
       WN_operator(this_wn) == OPR_MADD;
       this_wn = WN_kid0(this_wn)) {
    madd_count++;
  }

  //only one madd found
  if(madd_count <2)
    return wn;

  if(!WN_DP_Allowed(type))
    return wn;

  // transform MADDs to DP
  INT dp_count = madd_count / 2;
  INT segment_count = dp_count + 1;
  WN **segments = (WN **) alloca(segment_count * sizeof(WN *));
  memset(segments, 0, segment_count * sizeof(WN *));

  INT idx = 0;
  this_wn = wn;

  for(;idx<dp_count;idx++){
    WN *next_wn = WN_kid0(this_wn);
    WN *dp = WN_DP(WN_rtype(this_wn),WN_kid1(next_wn), WN_kid2(next_wn),
        WN_kid1(this_wn), WN_kid2(this_wn));
    segments[idx] = dp;
    WN_Delete(this_wn);
    this_wn = WN_kid0(next_wn);
  }

  segments[dp_count] = this_wn;

  // Add the partial sums.
  WN *tree = sum_madd_segments(segments, dp_count,0);
  return tree;
}
```

```
// step 3: Add the partial sums together
// The expr (r = a*b +..) is accumulated from left to right.
// (..((segments[high]+segments[high-1])+ segments[high -2])+..)+segments[0]
static WN *sum_madd_segments(WN **segments, INT high, INT low)
{
  WN *tree;
  if (high-low ==1) {
    WN *opnd1 = segments[high];
    WN *opnd0 = segments[low];
    tree = WN_Add(WN_rtype(opnd0), opnd1, opnd0);
  }else{
    WN *sum1 = sum_madd_segments(segments, high, low+1);
    WN *sum0 = segments[low];
    tree = WN_Add(WN_rtype(sum0), sum1, sum0);
  }
  return tree;
}
```

Selecting FSA at range propagation stage (§8.3.5, page 148).

```
static BOOL match_fsa_sequence(const RangeAnalysis &range_analysis, OP *op,
    OPS *ops)
{
  int val;
  TOP opcode = OP_code(op);
  if(opcode == TOP_call_i){
    TN *opnd_op = OP_opnd(op, 0);
    TN *opnd_r1 = OP_opnd(op, 1);
    TN *result = OP_result (op, 0);

    if(TN_is_symbol(opnd_op)){
      if(strcmp(ST_name(TN_var(opnd_op)),"__saddgs")==0){
        LRange_pc val1 = range_analysis.Get_Value(opnd_r1);
        LRange_pc rref = range_analysis.getLattice ()->makeRangeMinMax (0,0
            x7fffffff);

        if(rref->ContainsOrEqual(val1)){
          INT64 offset=TN_offset(opnd_op);
          mUINT8 relocs=TN_relocs(opnd_op);
          ST * st_org = TN_var(opnd_op);
          TY_IDX ty =ST_pu_type(st_org);

          ST * st=Gen_Intrinsic_Function(ty,"__sadds");
          TN * tn=Gen_Symbol_TN(st,offset,relocs);

          BB * bb = OP_bb(op);
          ANNOTATION *annot = ANNOT_Get(BB_annotations(bb), ANNOT_CALLINFO);
          CALLINFO *call_info = ANNOT_callinfo(annot);

          WN * call_wn = CALLINFO_call_wn(call_info);
          WN *new_call_wn = WN_CopyNode (call_wn);
          WN_st_idx(new_call_wn) = st->st_idx;

          CALLINFO_call_wn(call_info) = new_call_wn;
          CALLINFO_call_st(call_info) = st;

          PU& pu = Pu_Table[ST_pu (st)];
          Set_PU_is_pure (pu);
          Set_PU_no_side_effects (pu);

          Set_OP_opnd(op,0,tn);
          return TRUE;
        }
      }
    }
  }
  return FALSE;
}
```