N° d'ordre : 369
N° attribué par la bibliothèque : 06ENSL0369

THÈSE

en vue d'obtenir le grade de

Docteur de l'École Normale Supérieure de Lyon
spécialité: Informatique

Laboratoire de l'Informatique du Parallélisme

École doctorale de : Mathématiques et Informatique fondamentale

présentée et soutenue publiquement le 12 septembre 2006

par **Saurabh Kumar RAINA**

FLIP: A FLOATING-POINT LIBRARY FOR INTEGER PROCESSORS

Directeurs de thèse : Monsieur Claude-Pierre Jeannerod
Monsieur Jean-Michel Muller
Monsieur Arnaud Tisserand

Après avis de: Monsieur Jean-Claude Bajard
Monsieur Paolo Montuschi

Devant la commission d'examen formée de:

Monsieur Jean-Claude Bajard *Membre/Rapporteur*
Monsieur Christian Bertin *Membre*
Monsieur Bernard Goossens *Membre*
Monsieur Claude-Pierre Jeannerod *Membre/Directeur de thèse*
Monsieur Paolo Montuschi *Membre/Rapporteur*
Monsieur Jean-Michel Muller *Membre/Directeur de thèse*
Monsieur Arnaud Tisserand *Membre/Directeur de thèse*

Contents

Acknowledgments

This long voyage, from DEA to 12 September 2006 which marks the golden day of my life, wound through many sweet and bitter moments in these four years. It gives me immense pleasure to having received what I believe the finest education available. But all this could never have been possible without the assistance and cooperation of many people and organizations.

First and foremost, I would like to thank my advisors Claude-Pierre Jeannerod and Arnaud Tisserand without whom this thesis could never have seen this day. I thank them for their excellent supervision, technical discussions, providing me freedom and an almost independent atmosphere to work. I thank Claude-Pierre particularly for solving all my non-technical problems in a super-efficient way and for being a constant source of advice on organizing myself in order to survive and to come out triumphingly. He made my séjour in France merveilleux. I thank Arnaud for his advises of giving a professional touch to my research career. In particular, I really appreciate all those fruitful and versatile discussions courtes occasionnellement with him and thanks him for constant gruelling.

I also thank my jury members Jean-Claude Bajard, Paolo Montuschi and Bernard Goossens for accepting and reading such a heavy thesis, specially in vacations. Their comments, opinions and encouragement allowed me to improve the quality of this document. One again, thanks to all the jury members for making this possible.

I am immensely grateful to my Chief advisor Jean-Michel Muller, and Nicolas Brisebarre with whom it all began during the DEA internship and eventually I entered into the scientific research community.

I also thank my financial benefactors, la Région Rhône-Alpes and STMicroelectronics. I thank STMicroelectronicians, Christian Bertin for being one of the jury members, Hervé Knochel and Christoph Monat in particular for their highly spirited and motivated collaboration, for all kind of technical support, reading this document and last but not the least, great luncheons with the finest available vine. A special thanks to Hervé, with whom I developed a lively camaraderie, for listening to all my queries with patience and to be able to respond to all of them including the stupid ones too.

Now I thank all who have come across my research career and who have influenced my fundamentals of living a better life. A BIG thanks to Dear Arénairiennes: Jérémy, Nicolas Chantillon, Jean-Luc, Edouard, Guillaume, Romain, Francisco, Christoph, Pascal, David, Sylvie et Nicolas, Gilles and Nathalie for quick and funny discussions, Florent for Power PC and many other things, Serge for discussing just anything, Marc for introducing me to the IEEE standard. I will always remember and appreciate my friendship with Jer, Kiki, Jean-Luc and Ed. A special thanks to few Lipiens: Mme Boyer, Corinne, Dom, Nicolas, Antoine, Alain, Tanguy, Anne, Pushpinder, Victor, Stéphane.

Finally, I thank to my family and friends. I missed my parents presence on my defense day but fortunately my sister Chinu who is in France filled up this void. I would like to thank Manu whom I met very late during my PhD but who always stood stronger and longer besides me. Thanks Manu that you made such a short period so wonderful. I also thank Sualeh, Verma, Shrivastav,

Abdullah and Saif for their constant encouragement. My heart and my thanks goes to my mother Usha for her enduring patience and indefatigable support and my most special friend, colleague and mentor, Dhiraj.

Thanks LaTeX, Emacs, Debian and the Free Software Foundation.

Introduction

In this microprocessor dominated information world, the embedded systems have carved their own niche principally due to two reasons: they have a dedicated microprocessor for each specific task in many different application areas, so they are ubiquitous and they are cost and energy efficient.

Many multimedia applications doing scientific computations and/or digital signal processing depend on floating-point arithmetic due to high demands for accuracy. The performance and integrity of such applications can be directly linked with the question: "How efficiently and correctly can we perform floating-point arithmetic?"

General purpose processors may satisfy the hunger of these compute intensive applications but are not cost effective since they are not dedicated for a particular task. There exist embedded systems which have a dedicated floating-point unit but it comes at the cost of extra silicon area and increased power consumption. So, most embedded systems rely on *integer* or *fixed-point processors* and perform floating-point arithmetic in software. Generally, the software implementation of floating-point algorithms is optimized for a particular processor and probably for a particular application. But in most current software libraries the implementation is a direct mapping of the one that are used in the general purpose processors.

This thesis addresses the design and implementation issues involved in providing a *software layer* to support floating-point arithmetic on integer processors. There are many choices that need to be addressed, among which: internal representation of floating-point numbers, compliancy level with the IEEE standard (support of subnormal numbers or not), search for division and square root algorithms which can be well adapted to a particular processor, search for feasible range reduction methods for elementary functions. These choices influence the performance (latency, code size and throughput) of the software layer and eventually affect the performance of the applications relying on this layer. In spite of that, we are not aware of any literature which discusses these issues openly and broadly in the context of *embedded systems* to date. In most cases, the implementation provided in software is more or less the same as the one in hardware. A software implementation is probably optimized at the architectural level and might be targeted towards a bunch of applications but no efforts are made to do optimizations at the algorithmic level. This work discusses the possibilities of adequation and adaptation of specific algorithms to the ST200 family of processors from STMicroelectronics. It shows that the performance of a software implementation can be significantly improved through a sound understanding of a particular architecture, careful selection of an algorithm and a fine tuning of an implementation.

The foundation of this work was laid by STMicroelectronics who has developed a software library to support floating-point arithmetic on the ST200 family. The ST200 family consists of integer processors for embedded systems. This original library supports the basic arithmetic operations without supporting subnormal numbers and for rounding to nearest only. STMicroelectronics has already optimized this library for the ST200 family of processors.

Our software library FLIP, developed in a collaborative project funded by Région Rhône-Alpes and STMicroelectronics, provides support for floating-point arithmetic on two processors of the ST200 family, the ST220 and the ST231. The library has been optimized for these two target

processors. FLIP supports the following arithmetic operations grouped into three categories as:
Basic operations: $+, -, \times, /, \sqrt{}$, different levels of compliance to the IEEE standard.
Additional operations: $/, \sqrt{}, x^2, xy \pm z, 1/x, 1/\sqrt{x}$, fast implementation of division and square root, no subnormal support and rounding to nearest only.
Elementary functions: $\exp(x), \log_2(x), \sin(x), \cos(x)$, no subnormal support and rounding to nearest, for ST231 only.

A high-level validation of FLIP ensures its integrity and validates the overall numerical behavior of all the implemented algorithms. FLIP has achieved better performances in comparison to the original library, which – as said before – is already optimized. For operations such as addition/subtraction and multiplication a speed-up factor of 1.2 has been obtained. For complex operations like division and square root, FLIP is at least three times faster, and for operations such as inverse square root, a four fold increase in the performance has been achieved.

Regarding the implementation of elementary functions, the original library relies on FDlibm (a C math library) from Sun Microsystems. This library has not been optimized for any particular embedded processor and all the operations are performed directly on floating-point numbers. It assumes the presence of a floating-point unit on the machine that is used. So, for computing elementary functions, the original library for basic arithmetic operations is needed. For example, the additions and multiplications in the computation of $\sin(x)$ are perfomed on floating-point operands and thus a floating-point support for these operations is required. Due to this the latency of elementary functions ($\sin(x)$, $\cos(x)$, 2^x and $\log_2(x)$ for the moment) as measured on the target processors is high.

FLIP provides support for elementary functions by emulating all the floating-point operations involved as integer operations. So, FLIP does not rely on any other library; instead it interacts directly with the processor and functions at a lower level than FDlibm does. Nevertheless, we have measured the performances in three different ways as:

1. FDlibm with the original library;
2. FDlibm with FLIP for basic and additional operations;
3. FLIP for elementary functions.

This thesis is composed of four parts as follows:

Part I

- This part provides a general overview of the thesis.
- It recalls the *IEEE floating-point standard* for the single precision and presents the way of representing floating-point numbers and computing with them in the context of FLIP.
- It gives an overview of the *ST200* family of processors, the *ST200 compiler* and a presentation of the toolset. This helps to understand how a particular code is simulated, how its performance is measured and how it can be improved.
- It presents the *FLIP* library.
- It presents the different levels of *validation* of FLIP. Also presents our workbench which is composed of some *numerical applications*, some *industrial applications* and a compute intensive *computer game* and the achieved *performances*.

Part II

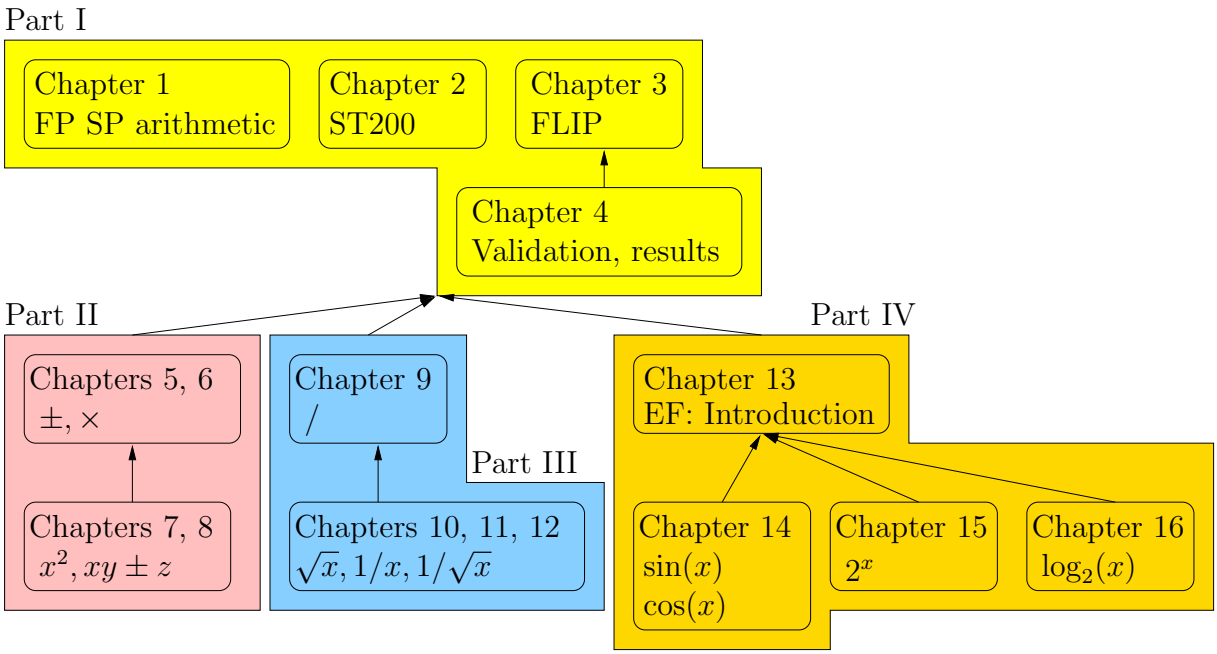


Figure 1: Organization of the thesis; FP: floating-point, SP: single precision, EF: elementary functions. Arrows signify the dependency of one chapter on another.

- This part presents the implementation of two basic arithmetic operations, *addition/subtraction* and *multiplication*. Recalls the steps of their basic algorithms and presents optimizations applied to some of these steps.

- It also presents the implementation of two other operations, *square* and *fused multiply-and-add*, based on the above two basic operations.

Part III

- This part can be considered as the most important part of this thesis.

- It recalls different classes of *division* algorithms and presents an extensive discussion about adapting of these algorithms to our target processors. Moreover, the chapter on division presents the content that will be useful to understand other operations presented in this part.

- Finally, this part provides implementation of *square root* and derived operations such as *reciprocal* and *inverse square root*.

Part IV

- This last part first introduces elementary functions, recalls some range reduction methods and an overview of the FDlibm library.

- It provides implementation of $\sin(x)$ and $\cos(x)$ for reduced range $[0, \pi/4]$ adapted to our target processors.

- It also provides implementation of 2^x and $\log_2(x)$ for all single precision floating-point numbers.

Apart from Part I, the order of presentation shows the chronological order of the realization of the work and the different versions of FLIP. Nevertheless there are some chapters presenting some operations which are dependent on the previously developed operations. The graph in Figure 1 gives a snapshot of the organization of this thesis as well as the dependencies between operations.

Part I

FLIP, a Floating-Point Library for Integer Processors

Chapter 1

Floating-Point Single Precision Arithmetic

Many scientific and engineering applications rely on numerical computation heavily. Since such applications demand a large range of representable numbers, almost all numerical computation uses floating-point arithmetic. The IEEE (Institute for Electrical and Electronics Engineers) published in 1985, the IEEE-754 standard for binary floating-point arithmetic. This standard specifies the representation of floating-point numbers as well as the behavior of the floating-point operations performed on these numbers.
It was followed two years after by the IEEE-854, radix-independent standard whose major goal was to deal with decimal arithmetic. This chapter can be divided into two parts. The first part presents the main foundation blocks which constitute the IEEE-754 standard. These are: 1) representation of floating-point numbers, 2) different floating-point formats, 3) representation of special values, 4) rounding modes, 5) exceptions, etc. The second part presents the representation of floating-point numbers in context of the FLIP library. Here we discuss, how FLIP represents floating-point numbers internally and performs computation with them.
This chapter equips the reader with the fundamental notions of floating-point arithmetic, particularly for single precision, and provides some definitions, terms and properties that will be used throughout this document. Finally, it introduces to the methodology of FLIP that is followed for implementing each floating-point operation.

1.1 Creation of the IEEE-754/854 Standard

In the early days of processor development each manufacturer developed its own floating-point system and thus it was natural to have different floating-point formats and different ways of doing floating-point computations on machines from different manufacturers. The behavior of the numerical programs was dictated by the architecture on which it was executed. Consequently, it was very difficult to write programs whose results were predictable, and which were portable, that is, which have a similar behavior when executed on different machines.

To come out of this predicament, in the late 1970s and early 1980s, from the strong inspira-

tion of the floating-point arithmetic of Intel's 8087 floating-point co-processor, the Institute for Electrical and Electronics Engineers proposed the IEEE-754 standard. The IEEE standard for binary floating-point arithmetic was published in 1985, when it became known officially as ANSI/IEEE Standard 754-1985 [14]. A second IEEE floating-point standard, for radix-independent floating-point arithmetic, ANSI/IEEE Standard 854-1987 [15], was adopted in 1987. The second standard was motivated by the existence of decimal, rather than binary, floating-point machines, particularly hand-held calculators, and set requirements for both binary and decimal floating-point arithmetic in a common framework. The demands for binary arithmetic imposed by IEEE 854 are consistent with those previously established by IEEE 754.

The IEEE standard defines the following points:

- Representation of floating-point numbers, formats and codings of special values.
- Some correctly rounded arithmetic operations such as $+, -, \times, /, \sqrt{}$, and remainder.
- Other operations such as comparisons and some conversions.
- Various rounding modes.
- Handling of exceptional situations such as division by zero.

This standard allows to preserve certain mathematical properties and thus maintains the quality of the result of a numerical program. The mathematical specifications of the standard are independent of the implementation problems, whether it is implemented in hardware by any machine or in software by any library. In this way it also allows the portability of programs.

1.2 Representation of IEEE Floating-Point Numbers

Floating-point representation is based on the familiar so-called *scientific notation* where nonzero numbers (in base 10) are expressed in a form like -7.6283×10^{-23}. Here, $-$ is the sign, 7.6283 is the mantissa, .6283 is the fraction, 10 is the radix and -23 is the exponent. For example, 5797.13 can be expressed in scientific notation by repeatedly dividing the mantissa by 10 until we get what is called a *normalized representation* of the number, that is, 5.79713×10^3. Here, the mantissa consists of a leading digit, to the left of the implied decimal point, and of a fraction to the right.

Similarly, in base 2, floating-point numbers are characterized by three fields, as shown in Figure 1.1: a sign s, an exponent e and a mantissa m.

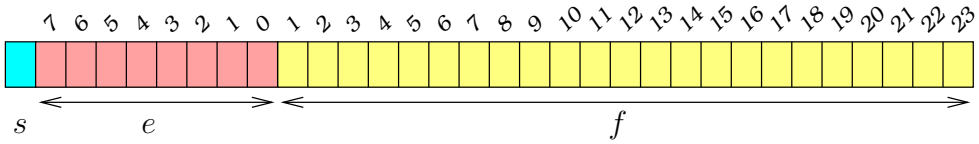


Figure 1.1: Three fields of a single precision floating-point number.

More precisely,

- The sign s, represented by one sign bit (sign-magnitude representation), indicates a positive ($s = 0$) or negative ($s = 1$) floating-point number.

- The exponent e (8 bits in the single precision format) is an integer defined by adding a *bias* to the mathematical value of the exponent; it is thus, called a biased exponent. For the single precision format, this bias is 127 and the range of the biased exponent is $[0\ (00000000)_2, 255\ (11111111)_2]$. For normalized numbers it lies between $e_{min} = 1$ and $e_{max} = 254$. The biased exponents $e_{min} - 1$ and $e_{max} + 1$ are reserved to represent special values (see Section 1.4).

- The mantissa $m = m_0.m_1m_2 \ldots m_{23}$ is a fixed-point number that consists of a leading digit (1 bit) to the left of the binary point and a fraction (23 bits) for the single precision format. The fractional part $f = m_1m_2 \ldots m_{23}$ is the internal representation of the mantissa and is called the *fraction* of the floating-point number. Since the first bit m_0 is always known, it is not necessary to store it and thus it is called the *implicit* (or hidden) bit. Two different types of mantissas exist:

  1. Normalized mantissa: the first bit m_0 is 1 and thus $m \in [1, 2)$.
  2. Denormalized (or subnormal) mantissa: the first bit m_0 is 0 and thus $m \in [0, 1)$. The value of the implicit bit is determined by the value of the biased exponent: $m_0 = 0$ if and only if $e = 0$.

Thus, the value of the floating-point number x in terms of its three fields s_x, e_x, m_x is

$$x = (-1)^{s_x} \times m_x \times 2^{e_x - 127}.$$

Example 1: The number $(1)_{10} = (1.00 \ldots 0)_2 \times 2^0$ has biased exponent $e = 0 + 127$ and fraction $f = 0$. Since $(127)_{10} = (01111111)_2$ and $(0)_{10} = (0)_2$, it is stored as:

0	01111111	00000000000000000000000

Example 2: The number $(-1/3)_{10} = (-0.333 \ldots) = (-1)^1 \times (1.0101 \ldots)_2 \times 2^{-2}$ has biased exponent $e = -2 + 127$ and fraction $f = .0101 \ldots$ which is a nonterminating binary number. Before the fraction is stored, it is *truncated* after 23 bits assuming the active rounding mode is round towards zero (see Section 1.5 for some other rounding modes). Since $(125)_{10} = (01111101)_2$ and the truncation of $(.\underbrace{0101 \ldots 010}101 \ldots)_2$ after 23 bits is $(.0101 \ldots 010)_2$, it is stored as:

1	01111101	01010101010101010101010

Since this type of representation of floating-point numbers uses an implicit bit, it is not possible to store zero. Thus, for zero (including the other special values), the standard provides a special representation (see Section 1.4).

1.3 IEEE Floating-Point Formats

The IEEE standard defines in fact four floating-point formats. The single precision format is 32 bits wide and the double precision requires 64 bits. There also exists an extended precision for these two formats. The width of all the components of a floating-point number in single and double precision formats is shown in Figure 1.2, and the complete characteristics of the single precision format are provided in Table 1.1. Figure 1.3 shows the different regions in which a

floating-point system divides the real numbers. This document deals with the implementation of IEEE *single precision* floating-point arithmetic only. For details on other formats, the reader is referred to [14, 70, 40, 26] (last two references are in french).

Precision	Sign	Exponent (biased)	Fraction
Single (32 bits)	1 bit	8 bits	23 bits
Double (64 bits)	1 bit	11 bits	52 bits

Figure 1.2: Widths of components in single and double precision IEEE format.

Single precision	
Base	2
Exponent	8 bits
Bias	127 ($= \frac{2^8}{2} - 1$)
Maximum exponent (e_{max})	
unbiased	127
biased	254
Minimum exponent (e_{min})	
unbiased	-126
biased	1
Reserved exponents	$e_{min} - 1$ and $e_{max} + 1$
Fraction/mantissa width	$23/23 + 1$ bits
Mantissa range	$[1, 2 - 2^{-23}]$
Largest normalized number (N_{max})	$(2 - 2^{-23}) \times 2^{127} \approx 3.4 \times 10^{38}$
Smallest normalized number (N_{min})	$2^{-126} \approx 1.2 \times 10^{-38}$

Table 1.1: Characteristics of single precision IEEE format.

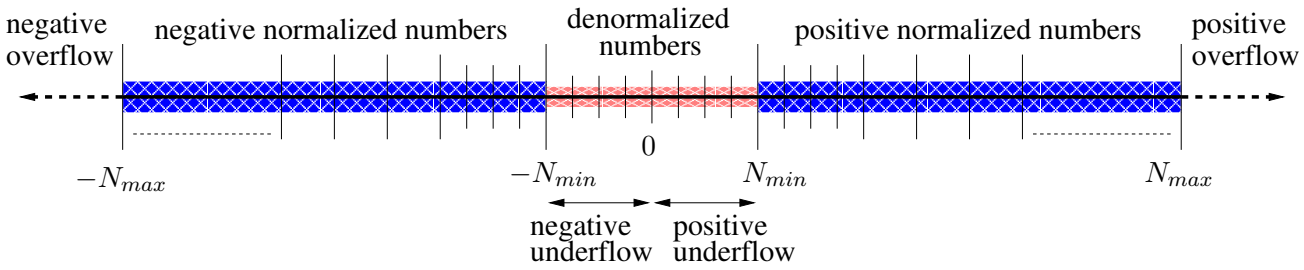


Figure 1.3: Different regions of the floating-point system on the real line.

1.4 Representable Numbers and Special Values

Table 1.2 provides the encoding of different representable numbers and special values in single precision.

Normalized numbers

As said before, they are represented by $x = (-1)^{s_x} \times 1.f_x \times 2^{e_x - 127}$ where $1 \leq e_x \leq 254$ and $m_x = 1 + f_x \in [1, 2)$.

Subnormal numbers

The subnormal (denormalized) numbers have $e_x = 0$ and a nonzero fraction (if the fraction is zero too, x would be the floating-point number zero). In this case, $m_x = 0.f_x \in [0, 1)$ and the exponent $e_x = 0$ when subnormal numbers are stored but is set to $e_{min} = 1$ while doing computation. Subnormal numbers are represented by $x = (-1)^{s_x} \times 0.f_x \times 2^{-126}$.

Signed zero

Due to the implicit leading one in the mantissa, the zero value is not representable by a normalized floating-point number. Zero is thus the special value such that with $e_x = f_x = 0$. The unspecified sign bit allows two values for zero: -0 and $+0$. This is consistent with the following properties : $\frac{1}{+0} = +\infty$ and $\frac{1}{-0} = -\infty$. The test $-0 = +0$ always returns a true value.

Not a Number

The value NaN(*Not a Number*) is encoded with $e_x = 255$ and $f_x \neq 0$. It is used to represent the value of an invalid operation such as $0/0$, $\sqrt{-1}$ or $0 \times \infty$. There exist two types of NaN: signaling NaN (sNaN) and quiet NaN (qNaN), each having more than one machine representation. Since the applications based on FLIP do not require two different NaNs, we shall treat both sNaN and qNaN as NaN in our work.

Infinity

Infinity values are encoded with $e_x = 255$ and $f_x = 0$. The sign bit distinguishes between $-\infty$ and $+\infty$. These values are used for overflows.

The type of representation of floating-point numbers provided by the IEEE standard facilitates the following operations:

1. Computation of the successor and the predecessor of a floating-point number;
2. Comparison of two floating-point numbers;
3. Detection of zero.

We shall now illustrate each of the three points above.

Computation of the successor and the predecessor of a floating-point number:

If x is a single precision floating-point number (not representing any ∞ or NaN) and X is an integer (signed) corresponding to the binary representation of x then the floating-point successor of x can be computed as

$$x^+ \leftarrow \begin{cases} X + 1 & x \geq +0, \\ X - 1 & x < -0, \end{cases} \tag{1.1}$$

	s_x	e_x	f_x
$1\sqrt{2}$	0	01111111	01101010000010011110011
2	0	10000000	00000000000000000000000
-3.25	1	10000000	10100000000000000000000
π	0	10000000	10010010000111111011011
1 000	0	10001000	11110100000000000000000
0.001	0	01110101	00000110001001001101111

	s_x	e_x	f_x
-0	1	00000000	00000000000000000000000
$+0$	0	00000000	00000000000000000000000
$-\infty$	1	11111111	00000000000000000000000
$+\infty$	0	11111111	00000000000000000000000
NaN	0	11111111	00000000000000000000001
NaN	0	11111111	11111111111111111111111

Table 1.2: Examples of representable numbers (above) and special values (below) in the single precision floating-point format.

and the floating-point predecessor of x as

$$x^- \leftarrow \begin{cases} X+1 & x \leq -0, \\ X-1 & x > +0, \end{cases} \tag{1.2}$$

where x^-, x, x^+ are three consecutive machine numbers (see Section 1.5 for a definition of machine numbers) such that $x^- < x < x^+$. In order to explain this property, first we present an example, given in Figure 1.4, of constructing a floating-point number from its binary (or hexadecimal) representation. So, x can be interpreted in three different bases as

$$\begin{aligned} X &= (\text{0xC7960000})_{16}, \\ x &= (-1.001011\ldots00 \times 2^{16})_2, \\ x &= (-76800)_{10}. \end{aligned}$$

Now we provide two examples to compute the floating-point successor and predecessor of x by doing a simple 2*'s complement* integer addition.

Example 3: Here x (X = 0x007FFFFF) is the largest positive subnormal number which is incremented to obtain its successor, that is, the smallest positive normalized number. Please note that 1 is represented as a 32-bit integer and not as a floating-point number. Hence, there are no vertical bars which are used to distinguish between three fields of a floating-point number.

	X	0	00000000	11111111111111111111111	$x = 0.11\ldots1 \times 2^{-126}$
+	1	0	00000000	00000000000000000000001	
		0	00000001	00000000000000000000000	$x^+ = 2^{-126}$

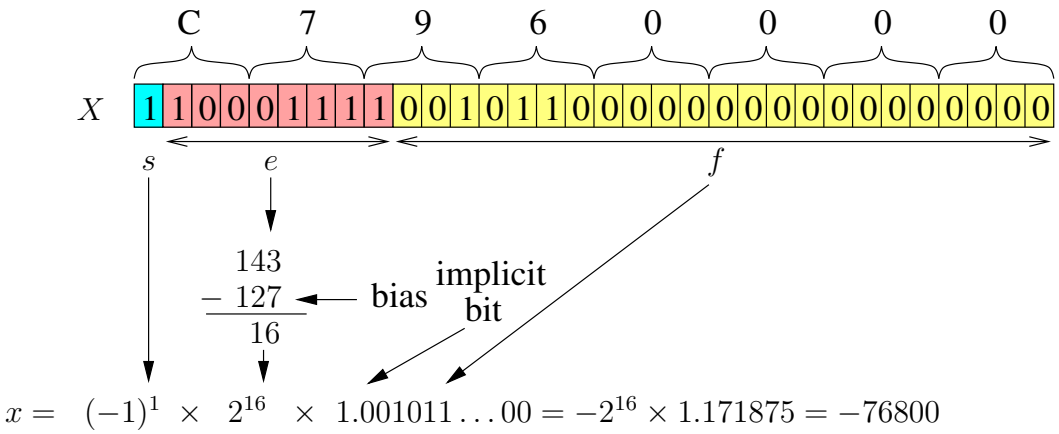


Figure 1.4: Constructing a floating-point number from its hexadecimal representation.

Example 4: Here x (X = 0x80000000) is negative zero and to obtain its predecessor, that is, the smallest negative subnormal number, X is decremented by adding 1 to it. Indeed, this addition will produce an integer larger than X in 2's complement representation but will produce a floating-point value (*sign-and-magnitude*) smaller than x.

	X	1	00000000	00000000000000000000000	$x = -0$
+	1	0	00000000	00000000000000000000001	
		1	00000000	00000000000000000000001	$x^- = -2^{-149}$

This property allows us to compute the successor and/or the predecessor of a particular floating-point number by performing a simple addition which can be easily performed on integer processors. We use this property very frequently for various optimizations in our FLIP library.

Comparison of two floating-point numbers:

It can be easily inferred from the above property of computation of successor and/or predecessor that if X and Y are the integers corresponding to the binary representation of two distinct positive floating-point numbers x and y then $X < Y \iff x < y$. Thus, the magnitudes of two floating-point numbers can be compared as if these numbers were integers.

Detection of zero:

The zero floating-point number operand has both the exponent and the mantissa fields equal to zero, while the sign bit determines its sign (± 0). The integer corresponding to this zero floating-point operand is shifted left by 1 bit in order to get rid of the sign bit. The magnitude of the operand is compared with the integer representation of 0 that allows to detect whether a given floating-point number is equal to zero.

1.5 IEEE Rounding Modes

It is not possible to represent all the real numbers in any given floating-point format, and especially in the single precision format. Hence one defines a *machine number* to be a number that can be exactly represented in the target floating-point format. The exact result $\hat{z}$ of an arithmetic operation whose operands are machine numbers may not be representable in the target format (e.g. $(1/3)_{10} = 0.333\ldots$ or $(1/10)_2 = 0.000110\ldots$). The exact result must be *rounded* before it is stored in the target format. This causes an error that is called a rounding error. In a floating-point system that follows the IEEE standard, the user can choose an *active rounding mode* among the four rounding modes provided by the standard which are as follows:

- rounding towards $+\infty$: denoted by $\triangle(\hat{z})$, returns the smallest machine number greater than or equal to the exact result $\hat{z}$.
- rounding towards $-\infty$: denoted by $\nabla(\hat{z})$, returns the largest machine number smaller than or equal to the exact result $\hat{z}$.
- rounding towards 0: denoted by $\mathcal{Z}(\hat{z})$, returns $\triangle(\hat{z})$ for negative numbers ($\hat{z} < 0$) and $\nabla(\hat{z})$ for nonnegative numbers ($\hat{z} \geq 0$).
- round to *nearest*: denoted by $\circ(\hat{z})$, returns the machine number that is the closest to the exact result $\hat{z}$. If $\hat{z}$ is the exact middle between two consecutive machine numbers, it returns the machine number whose least significant bit is 0 (*round-to-nearest-even*). The reasons for preferring an even machine number to an odd one can be consulted in [59, p.236]. This is the default rounding mode in practice.

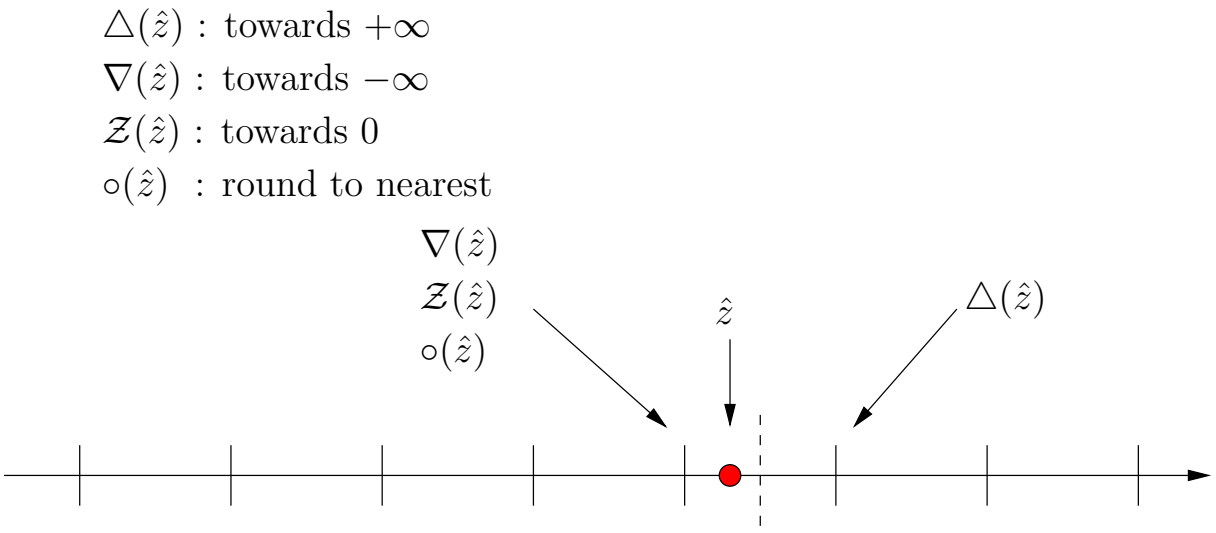


Figure 1.5: Different possible roundings of the exact result $\hat{z}$ (here $\hat{z} > 0$).

The four rounding modes are illustrated in Figure 1.5. The IEEE standard requires the following property for arithmetic operations.

Property 1.5.1 (Correct rounding) *Let x and y be two machine numbers, $\circledcirc$ be any of the four basic arithmetic operations $\{+, -, \times, /\}$ and $\diamond \in \{\triangle, \nabla, \mathcal{Z}, \circ\}$ be the active rounding mode among the four IEEE rounding modes. The computed result of $x \circledcirc y$ must be $\diamond(x \circledcirc y)$, that is, the system must behave as if the result were first computed* exactly*, with infinite precision, and then rounded.*

This property is called *correct rounding* and the operations that satisfy this property are called *correctly (or exactly) rounded*. The IEEE standard specifies this property for addition, subtraction, multiplication, division, remainder and square root. Unfortunately, there is no similar specification for elementary functions in which case we will use the notion of *faithful rounding* [27].

Definition 1.5.1 (Faithful rounding) *The faithful rounding of a real number gives either of the two machine numbers which surrounds the real number, and the machine number itself if it is equal to the real number.*

The error committed during rounding can be measured in different ways. The approximation of the exact result $\hat{z}$ by a machine number z can be measured in absolute error, relative error and/or number of ulps.

Definition 1.5.2 (Absolute error) *The absolute error E_{abs} is the distance between the exact result $\hat{z}$ and its machine representation z:*

$$E_{abs} = |\hat{z} - z|.$$

Definition 1.5.3 (Relative error) *The relative error E_{rel} is the difference between the exact result $\hat{z}$ and its approximation z, divided by the exact result $\hat{z}$:*

$$E_{rel} = \frac{|\hat{z} - z|}{|\hat{z}|} = \frac{E_{abs}}{|\hat{z}|}.$$

The term ulp (unit in the last place) was coined by W. Kahan in 1960 [58] and is frequently used for expressing errors in floating-point computations.

Definition 1.5.4 (Ulp) *If $\hat{z}$ is not a floating-point number then ulp($\hat{z}$) is defined as $\triangle(\hat{z}) - \nabla(\hat{z})$. In case of floating-point numbers, let us say if $\hat{z}$ is rounded by a floating-point number z, then ulp(z) = $|z^+ - z|$ where z, z^+ are two consecutive floating-point numbers such that $z < z^+$. Moreover, ulp(NaN) is NaN.*

The notion of ulp has been illustrated in Figure 1.6. It shows the computation of ulp of different numbers. Also, the rounding error, when $\hat{z}$ is approximated by z, expressed in number of ulps is ≤ 0.5 ulp($\hat{z}$) for round to nearest and < 1 ulp($\hat{z}$) for round towards 0, $+\infty$ and $-\infty$. When z changes from a power of two to the next smaller floating-point number, ulp(z) decreases by a factor of two. The powers of two divide the floating-point system into different *binades*. Floating-point numbers in the binade $[1, 2)$ are spaced half as far apart as floating-point numbers in the binade $[2, 4)$. Hence, ulp(z) vary by a factor of two in two consecutive binades. A very fruitful discussion about the notion of ulp has been presented in [67].

1.6 Exceptions

Exceptions play an important role by maintaining a chronicle of exceptional events and handling them. Handling of such events is done by issuing default responses like setting flags or by acting according to the user-defined traps. Exceptions help users to minimize the complications arising from exceptional situations by informing about the behavior of an operation and its result.

The standard anticipates that the arithmetic system continues to perform computation as long as possible, handling exceptional situations by setting appropriate flags. There are five types of exceptions (Invalid, division by zero, overflow, underflow and inexact) that shall be signaled when detected. The signaling involves setting a status flag, taking a user-defined trap, or possibly doing

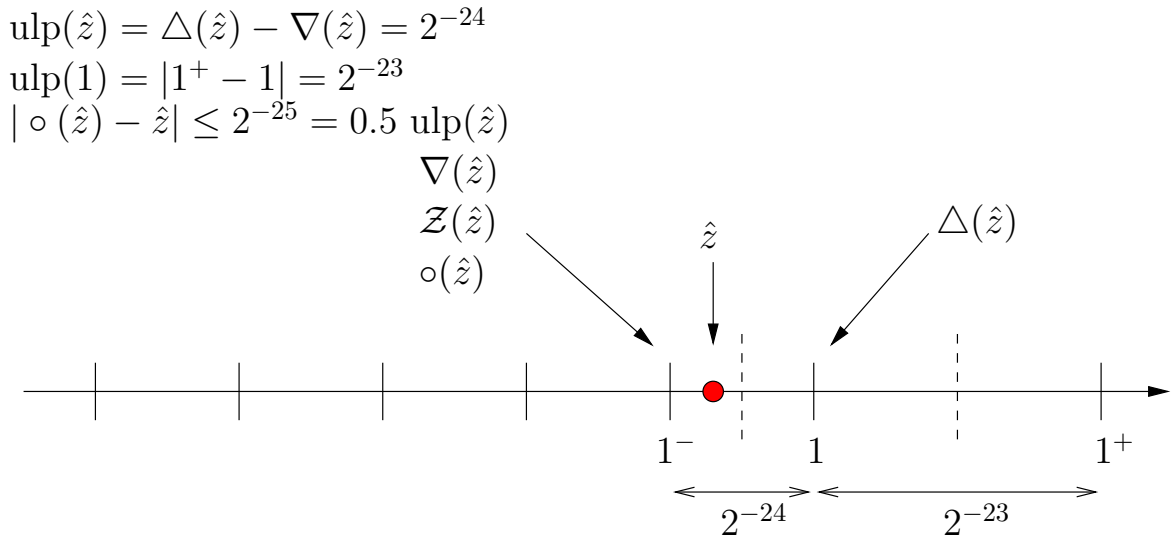


Figure 1.6: Illustration of ulp of different numbers and expressing rounding error in terms of ulps.

both. These flags are called *sticky* flags, that is, once they are set they stay in the set state until reset by the user. Resetting a flag clears it, and it stays clear until the first occurrence of an exceptional event that sets it again, or until the user restores it to a previous state. The user can set or reset these status flags individually or all the five at one time.

The only exceptions that can simultaneously be set are inexact with overflow, and inexact with underflow.

Invalid Operation

If an operand is invalid for the operation to be performed, the invalid operation exception shall be signaled and a NaN will be returned as a result. If the result of an operation is NaN because one of the input operand is NaN then no exception will be signaled. The invalid operations are, among others:

1. Addition or subtraction: subtraction of infinities like $(+\infty) + (-\infty)$;
2. Multiplication: $0 \times \infty$;
3. Division: $0/0$ or ∞/∞;
4. Remainder: $x\%y$, where y is zero or x is infinite;
5. Square root if the operand is less than zero, for example $\sqrt{-5}$. The standard maintains that $\sqrt{-0} = -0$.

Division by Zero

If the divisor is zero and the dividend is a finite nonzero number, then the division by zero exception shall be signaled. The result shall be $+\infty$ or $-\infty$ depending on the sign of the operands. The division by zero flag, when set, indicates precisely that an infinite result was delivered by some previous floating-point operation on finite operands, that the exact result is not finite. For example, $x/0$ with $x \notin \{0, \infty, NaN\}$. No exception is signaled when an infinite result is produced from infinite operands, as in $\infty \times \infty$, $\infty/0$, $\sqrt{+\infty}$.

Overflow

The overflow exception shall be signaled when the exact result of an operation is finite but can not be stored in the given format since its absolute value exceeds N_{max}, the largest finite floating-point number. For example, $N_{max} + 1$ and $\exp(10^3)$. The result shall be determined by the rounding mode and the sign of the intermediate result, as illustrated in the following table.

Rounding mode	Sign of the intermediate result	Returned result
towards $+\infty$	$+$	$+\infty$
	$-$	$-N_{max}$
towards $-\infty$	$+$	N_{max}
	$-$	$-\infty$
towards 0	$+$	N_{max}
	$-$	$-N_{max}$
to nearest	$+$	$+\infty$
	$-$	$-\infty$

The overflow flag, when set, indicates precisely that an *infinite* result was delivered by some previous floating-point operation on finite operands, but the *exact* result is *finite*.

Underflow

The underflow shall be signaled when the exact result of an operation lies in the interval $(-N_{min}, N_{min})$ (see Figure 1.3), that is, its magnitude is smaller than the smallest normalized floating-point number. For example, $N_{min}/3$ or $\exp(-137)$. The returned result might be 0, a subnormal, or $\pm N_{min}$.

Inexact

As the name says, if the exact result of an operation can not be represented exactly in the given floating-point format and thus requires rounding, the inexact exception shall be signaled. For example, $(1/3)_{10}$ or $(1/10)_2$.

1.7 FLIP Representation of Floating-point Numbers

Internal representation and *manipulation* of floating-point numbers using integers, *correct* and *efficient* implementation of floating-point algorithms based on this representation and then performing numerical computations that rely heavily on these algorithms is the major achievement of this research.

In order to explain better the optimization techniques employed in FLIP, and presented in this document, we adopt a way of representing floating-point numbers through binary strings. This representation allows to compute with the three fields of a floating-point number independently and simultaneously. It also facilitates the manipulation of *individual bits* of these fields which can be done almost all the time in parallel and thus allows to improve the overall efficiency.

We represent the three fields of a floating-point number through binary "strings" denoted by $\mathrm{S}, \mathrm{E}, \mathrm{M(F)}$ whose values are denoted by $s, e, m(f)$. A floating-point number is stored/packed using three strings $(\mathrm{S}, \mathrm{E}[7:0], \mathrm{F}[1:23])$ with sign bit $\mathrm{S} \in \{0,1\}$, exponent string $\mathrm{E}[7:0] \in \{0,1\}^8$ and fraction string $\mathrm{F}[1:23] \in \{0,1\}^{23}$. These strings concatenate and form a single precision

IEEE floating-point number stored on a 32-bit register as shown in Figure 1.7. Figure 1.1 shows the IEEE representation of floating-point numbers where the three fields are represented as three independent mathematical quantities; sign and exponent being an integer and fraction being a real quantity. Here, in Figure 1.7, all these fields are considered as integer strings of bits which is the internal representation of floating-point numbers adopted by FLIP.

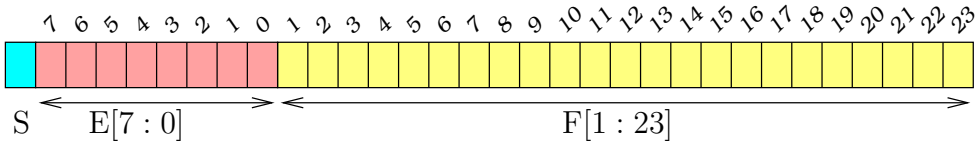


Figure 1.7: Encoding of a single precision (32 bits) floating-point operand.

The values of the exponent and fraction are given by

$$e = \sum_{i=0}^{7} \mathrm{E}[i]2^{i}, \qquad f = \sum_{i=1}^{23} \mathrm{F}[i]2^{-i},$$

and $s = \mathrm{S}$. In the above notation, $X[i]$ denotes the ith element of the sequence X as represented by X_i in mathematical text, or the ith element of the array $X[\ ]$ as represented in C language. Since computations are done with the implicit bit reinserted to the fraction string, that is, with the mantissa, we define the mantissa string as $\mathrm{M}[0:23] = \mathrm{M}[0].\mathrm{F}[1:23]$ and the value of the mantissa is given by

$$m = \sum_{i=0}^{23} \mathrm{M}[i]2^{-i},$$

where

$$\mathrm{M}[0] = 1 \quad \text{and} \quad m = 1.f \in [1, 2) \quad \text{for normalized numbers,}$$

and

$$\mathrm{M}[0] = 0 \quad \text{and} \quad m = 0.f \in [0, 1) \quad \text{for subnormal numbers.}$$

Although FLIP supports subnormal numbers for the basic arithmetic operations $(+, -, \times, /, \sqrt{})$, we consider all the floating-point numbers as normalized floating-point numbers except stated otherwise.

1.8 FLIP Methodology for Computing with Floating-Point Numbers

This section presents the computation of $z = \diamond(g(x, y))$ in context of FLIP. Here, g is a function implemented in FLIP for a particular floating-point operation and x, y, z are floating-point numbers. Computing directly with the floating-point numbers is not preferable due to their dynamic range and the complex scaling operations required in the algorithms based on them. So FLIP follows the common methodology, that is followed in other software libraries too, of implementing floating-point operations. Under this methodology the three fields of the input floating-point operand(s) are extracted and computations are performed on the values contained in these fields. To understand better this methodology a three-stage procedure is shown in Figure 1.8.

The first stage, called "UNPACK", extracts the exponent and the fraction part of each floating-point input operand and converts them into an internal format. They are stored as integers (signed

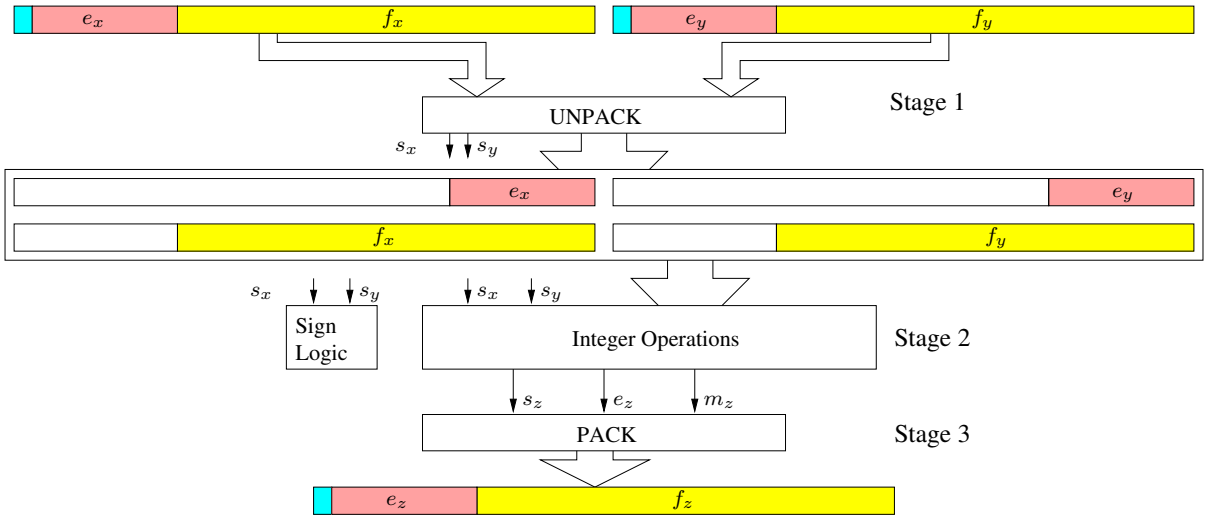


Figure 1.8: High-level description of implementation of any floating-point operation.

or unsigned) in two separate 32-bit registers. The second stage consists in sign logic and computing with these integers; this stage produces in the end, the three fields, sign s_z, exponent e_z (32 bits) and mantissa m_z (32 bits), of the result. The last stage, called "PACK", packs these three fields, masking the mantissa before it is stored so as to include only the fraction f_z, in a 32-bit register as a single precision floating-point number. Since for each floating-point operation the first and the last stages are the same and involve simple shifts and maskings, we shall discuss only the second stage for each operation in this document.

This methodology allows to compute with floating-point numbers in a fixed interval $[1, 2)$. In the second stage, we deal with either mantissas or floating-point numbers belonging to a fixed short interval that can be handled easily. Now we provide two examples which show how a particular value can possibly be represented in a register and how addition, multiplication, etc. of these values are emulated by integer addition, multiplication, shifting, etc.

Example 5: A real number $b = 3.7890627$ can be represented in a 32-bit register as

2^1	2^0	2^{-1}	2^{-2}	...	...	...	...	2^{-28}	2^{-29}	2^{-30}
1	1	1	1	0 0 1 0 1 0	...	...	...	0	1	1	1

.

The infinite binary expansion of b has been rounded, at the 30th fractional bit, to the nearest number. The long vertical bar indicates the position of the binary point. Initially, b is stored with the precision of 32 bits, the maximum available precision, with the leading bit 1. But it may not always be the case, as shown below when $b' = b - 3.7800002 = 0.0090625$ is stored in a 32-bit register as

2^{-1}	2^{-2}	...	2^{-6}	2^{-7}	...	...	...	...	2^{-30}	2^{-31}	2^{-32}
0	0	000	0	1	0	0101	0001	...	0	1	0	1

.

Here the infinite binary expansion of b' has been rounded at the 32nd fractional bit but is stored with six leading zeros which results in a *relatively* less accurate b'. Moreover, there is a big

difference between the representation and usage of a particular number. It highly depends on how a number is used for any computation and with which number. Let us say, when b is added with a larger number 37.89 having a different binade as shown below, then the binary point must be aligned for the addition to take place.

		2^5	...	2^1	2^0	2^{-1}	...	2^{-26}	2^{-27}	2^{-28}	2^{-29}	2^{-30}
	0.3789	1 0 0 1		0	1	1	1000...	... 1				
+	b	0 0 0 0		1	1	1	0010...	... 1	0	1	1	0

This alignment causes b to shift right which eventually causes loss of accuracy. In this case the representation of b will have four leading zeros. So, depending on the computation and the range of the numbers, the binary point may float to any position on a 32-bit register. Avoiding such situations where large alignment shifts are required is an important implementation issue. Practically, the numbers are well chosen, for example the coefficients in polynomial computation, so that they have almost a same binade. Also, knowing beforehand the type of computation that will be performed really helps a lot as shown below when b' is multiplied by 0.3789.

		2^{-2}	2^{-3}	...	...	...	2^{-32}	2^{-33}
	0.3789	1	1	000001	...	...10	0	1
		2^{-7}	2^{-8}	...	...	...	2^{-37}	2^{-38}
×	b'	1	0	010100	...	...10	0	0

It shows that in case of multiplication two numbers of different binade can be positioned in such a way that the leading bit is always 1 so as to have the maximum accuracy possible. Note that the weight of the leading bit for 0.3789 is 2^{-2} and for b' it is 2^{-7}. This helps to guess the binade of all the intermediate and final results.

Example 2: Let us explain with the help of this example; the steps involved in computing $a \times b + c$ where $a = 1.515625$, $b = 3.7890625$ and $c = 31.50390625$. Figure 1.9 below shows how a, b, c are represented in the best possible way so as to have the maximum accuracy. It also shows the alignment of the intermediate result for addition to take place.

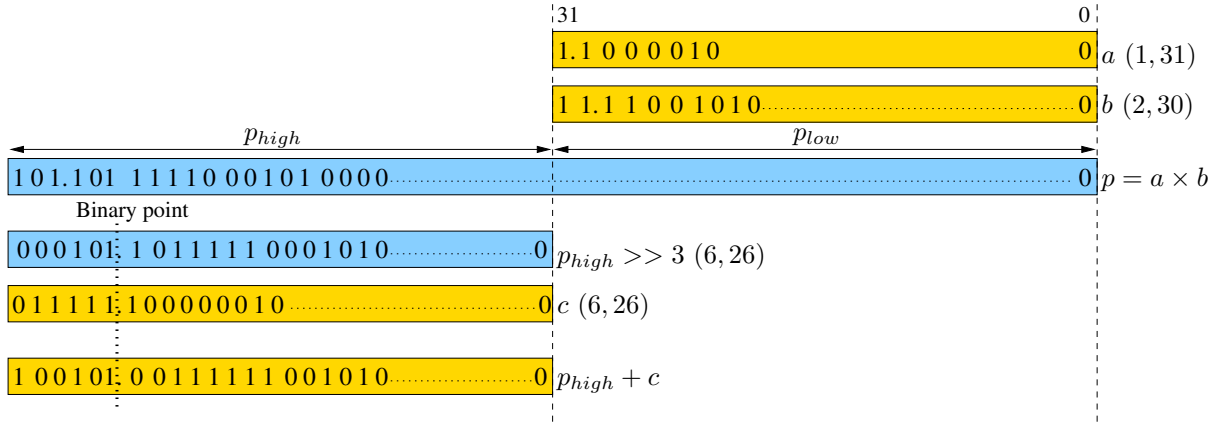


Figure 1.9: Illustration of the steps of computation when emulation is done.

All the real values are stored in a 32-bit register with the number of integral and fractional bits given in the parenthesis, for example, b $(2, 30)$ indicates that b has 2 integral bits and 30 fractional bits when stored in a 32-bit register as required by the type of computation. The advantage of using this notation is the ability to track how the binary point floats in the intermediate computations and finally provide a correct final result. The product $p = a \times b = 2^{32}p_{high} + 2^{32}p_{low}$ has 64 bits but the lower part p_{low} must be discarded and the computation is done with the most 32 significant bits. Notice that p_{high} is shifted right by 3 bits in order to align the binary point as required by the addition $p_{high} + c$. Also, c is stored in a 32-bit register with a leading 0 bit. It is required by this addition since $p_{high} + c$ might generate a carry. Hence, c $(6, 26)$ has 6 integral bits and 26 fractional bits when stored in a 32-bit register.

Chapter 2

Target Processors and Environment

A vast and in-depth knowledge of the target processor architecture and of the environment where they will be used has been a key factor in the development of the FLIP library. From the conception of the library to the evaluation of its performance, a thorough understanding of the target processors has influenced the implementation of floating-point operations at each stage and has helped making even the smallest decisions during the writing of programs. Also, without acquiring the skills to use the development tools provided by STMicroelectronics FLIP would have resulted in just a bunch of programs.
This chapter presents the ST200 family of processors and particularly the two target processors, ST220 and ST231. Depending on the characteristics of these processors, optimizations are done, at both algorithmic and implementation level. A brief overview of the available hardware resources, specification and latencies of the most useful instructions has been provided. This is followed by an overview of the ST200 compiler with different optimization options and a description of the set of supported intrinsic functions. In the end, a brief overview of different tools is provided.

2.1 ST200 Processor Cores

The ST200 family of high-performance low-power VLIW (see Section 2.1.1 for a definition of VLIW) processor cores are targeted at STMicroelectronics system-on-chip (SOC) solutions for use in computationally intensive applications as a host or an media processor. Such applications include embedded systems in consumer, digital TV and telecommunication markets where there is a specific need for high-performance low-cost audio/video data processing.

Applications for ST200 family-based processors can be developed in C or C++ in conjunction with the complete tool support for OS21 (small real-time operating system developed by STMicroelectronics) and Linux.

Unlike a super-scalar processor, the ST200 VLIW processors are significantly simpler and smaller. The compiler is key to generate, schedule and *bundle* (see Section 2.2.4) operations, removing the need to add complex hardware to achieve the same results.

The ST200 design originates from the Lx research project. This project was started as a joint development by Hewlett-Packard laboratories and STMicroelectronics [36]. According to Josh

Fisher who is often referred to as "the father of VLIW", the Lx architecture consists in a family of cores which has been designed for the embedded applications. STMicroelectronics purchased the rights to develop the Lx architecture [53] and to create the ST200 processor family.

In this document we consider only the ST220 and the ST231 cores of the ST200 family, for which FLIP was target. A complete description of the architecture and of the toolset of both processors are detailed in [88, 89, 90]. Most of the information about the architecture and the toolset for ST220 and ST231 provided in this chapter is inspired from these three manuals.

This ST200 family of embedded processors uses a scalable technology that allows variation in instruction issue width, the number and capabilities of functional units and register files, and the instruction set. The principal features of the ST200 family are as follows:

- Parallel execution units, including multiple integer ALUs[1] and multipliers.
- Architectural support for data prefetch.
- Predicated execution through select operations.
- Efficient branch architecture with multiple condition registers.
- Encoding of immediate operands up to 32 bits.

2.1.1 What is VLIW?

VLIW (for Very Large Instruction Word) processors use a technique where instruction level parallelism is explicitly exposed to the compiler which must schedule operations to account for the operation latency.

RISC[2]-like instructions (syllables) are grouped into bundles (wide words) and are issued simultaneously. While the delay between issue and completion is the same for all instructions, some results are available for bypassing to subsequent instructions prior to completion. This is discussed further in Section 2.2.2.

A VLIW processor is significantly simpler than a corresponding multiple issue superscalar processor since the complexity instruction grouping and scheduling hardware has been reduced by moving this to the instruction scheduling system (compiler and assembler) in the software toolchain.

2.1.2 From ST220 to ST231: Overview and Differences

The ST220 and ST231 are a single cluster member of the ST200 family. Figure 2.1 displays the basic organization of the ST200 family processor.

Clustering is a technique in which functional units and registers are tightly coupled in groups called clusters.

The ST220/ST231 cluster consists of functional units and two register files (branch registers and general purpose registers) but does not include the instruction fetch mechanism, caches or control registers. Inside the functional core of the processor there exist execution units working on two register files. These execution units include four integer units, two multiply units, a load/store unit and a branch unit which are described in Section 2.2.1.

In comparison to the ST220, the ST231 core has lower power consumption and improved multiply units. Operation latency interlocks have been added in order to reduce the number of

[1] Arithmetic and Logic Unit

[2] Reduced Instruction Computer Set

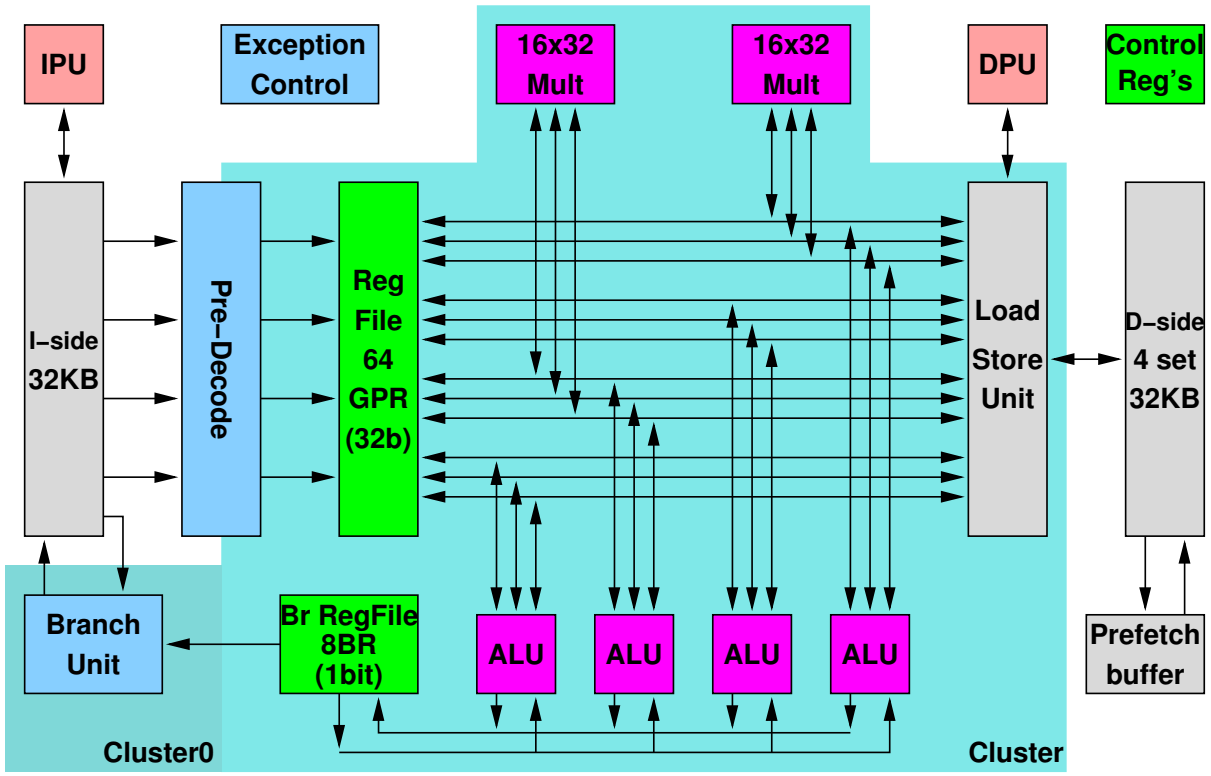


Figure 2.1: The ST200 VLIW processor architecture (here ST220).

required *nops* (no operations), and therefore reduce the code size. A number of new 32×32 multiply instructions, `mul32`, `mul64h`, `mul64hu` and `mulfrac` have been added. These instructions help to improve the run-time of some algorithms (implemented in FLIP) which require full-width multiplication (32×32).

2.2 Hardware Resources

This section describes the following in context to both the processors:

1. The execution units including four integer units, two multiply units, a load/store unit and a branch unit.
2. Register files.
3. Execution pipeline and latencies for various instructions.
4. The ST220/ST231 processor memory subsystem.
5. Description of all the instructions in the ST220/ST231 instruction set and bundle encoding.

From now on, the name of the family "ST200" will be used to identify both the target processors (ST220 and ST231) in this document. Individual names will be used to specify the differences.

2.2.1 Execution Units

Apart from the instructions available for integer addition, subtraction and multiplication, there exists a division step instruction performing one step of the nonrestoring division algorithm and

allowing to produce a quotient of up to 32 bits. In order to reduce the use of conditional branches, the ST200 processor also provides conditional selection instructions.

Integer Units

The ST200-based processors have four identical integer units. Each integer unit is capable of executing one arithmetic or logic operation per cycle. The results of the integer units can be used as operands of the next bundle. This is equivalent to a pipeline depth of one cycle.

Multiply Units

The ST200 has two identical multiply units. Each multiply unit is pipelined with a depth of three cycles, starting an operation every cycle. On the ST220 core, each multiply unit takes a 16-bit and a 32-bit integer operands and produces a single 32-bit result ($16 \times 32 \rightarrow 32$-bit), but the width of operands have been extended on the ST231 core. There exists a $32 \times 32 \rightarrow 32$-bit multiplier which takes two 32-bit integer or fractional operands and produces a 32-bit result.

This enhancement of multiplier gives ST231 an edge over ST220, for example, while multiplying two 24-bit mantissas the ST231 needs 3 cycles whereas the ST220 requires 6 cycles.

Load/store Unit

The ST200 has a single load/store unit. The load/store unit is pipelined with a depth of three cycles, executing an instruction every cycle. Uncached accesses or accesses which miss the data cache cause the load/store unit to stall the pipeline to ensure correct operation.

Dismissible loads are used to support software load speculation. This allows the compiler to schedule a load in advance of a condition that predicates its use.

Branch Unit

The ST200 has one branch unit. There is a support for both relative immediate branches (with and without condition code), and absolute and relative jumps and calls.

A conditional branch is performed using the `br`, `brf` and `goto` (immediate) instructions. An unconditional jump is performed using the `goto` (link register LR) instruction. An unconditional call is performed using the `call` (link register) and the `call` (immediate) instructions. Due to pipeline restrictions, all branches and jumps incur a penalty of one cycle of stall. So an extensive use of such instructions in a program may damage its performance. In our FLIP library, we have taken special care of comparisons resulting in such branches and jumps in order to avoid them as much as possible.

Register Files

The general purpose register file contains 64 words R0, ..., R63 of 32 bits each. Reading register R0 always returns the value zero and writing to R0 has no effect on the processor state. The branch register file contains eight single-bit-wide registers used for controlling conditional branches; they are called B0, ..., B7.

2.2.2 Execution Pipeline and Latencies

Execution Pipeline

The ST200 features a clean, compiler-friendly 6-stage instruction pipeline depicted in Figure 2.2, where all instructions are fully pipelined. The *execution pipeline* is three cycles long and comprises of three stages E1, E2 and E3. All instructions begin in E1. All results are written at the end of E3. The result(s) of the instruction which complete earlier than E3 are made available for bypassing as operands to subsequent instructions.

Instructions Latencies

ST200 instructions begin in E1 stage and complete in either E1, E2 or E3 (or write phase). The time taken for an instruction to produce a result is called the instruction latency. This processor executes up to four instructions per cycle, with a maximum of one control instruction (goto, jump, call, return), one memory instruction (load, store, prefetch), four ALU instructions and two multiply instructions per cycle.

Almost all instruction latencies are 1 cycle, meaning that the result of an instruction can be used in the next bundle. Some instructions have a 3-cycle latency (load, multiply, compare to branch) or in one case a 4-cycle latency (load link register to call).

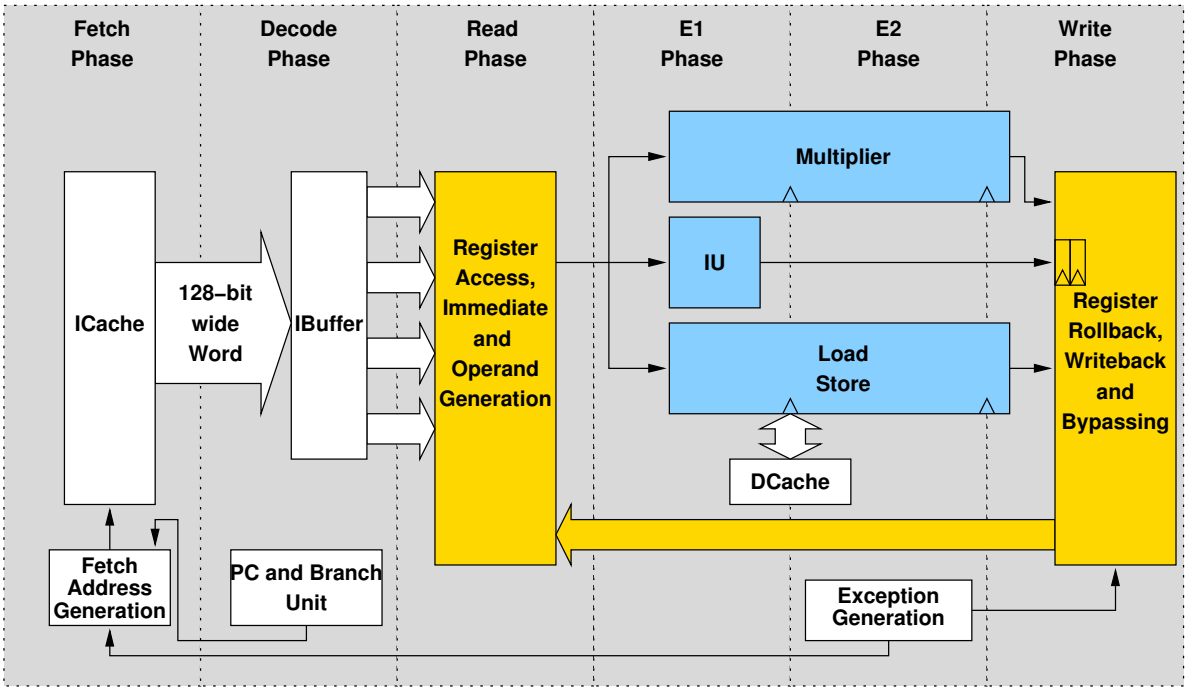


Figure 2.2: The ST200 processor pipeline.

2.2.3 Memory Subsystem

The memory subsystem includes the caches, protection units, write buffer, prefetch cache and the core memory controller (CMC). It is split broadly into two sides, the instruction side (I-side) and the data side (D-side) (see Figure 2.1). The I-side, containing the instruction cache, supports the fetching of instructions. The D-side, containing the data cache, prefetch cache and write buffer, supports the storing and loading of data.

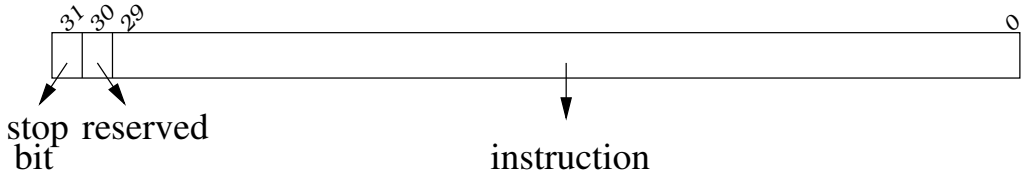


Figure 2.3: Encoding of a syllable.

I-side Memory Subsystem

The instruction buffer issues instructions to the processor core. It attempts to fetch ahead in the instruction stream in order to keep its buffer full. In case of branch, it will take one cycle to fetch the next bundle from the cache; this means the processor will stall for one cycle. If the branch is to a bundle that spans over two cache lines then it will take two cycles to fetch the bundle and thus the processor will stall for two cycles.

When instructions are requested from the cache, the instruction buffer verifies whether they are already present in the cache. If it is not the case, they are fetched from memory and stored into the cache, during which time the processor will stall. This results in an instruction cache miss. The instruction cache is a 32-K direct mapped cache.

D-side Memory Subsystem

All data accesses take place through the D-side memory subsystem. The data cache is 32 Kbyte 4-way associative with a 32-byte line (see [50, p. 392] to know more about cache associativity).

The load store unit (LSU) performs all data access operations. The prefetch cache prefetches and stores data from external memory and sends it to the data cache when (and if) it is required.

If the required data is found in the prefetch cache, it is loaded into the data cache as if it were fetched from external memory. If the required data is not found in the prefetch cache, the data cache stalls until the data is returned from external memory. This results in a data cache miss.

The prefetch cache explicitly fetches data from external memory but hides the latency associated with that fetch and thus improves the performance.

2.2.4 The ST200 Integer Instruction Set

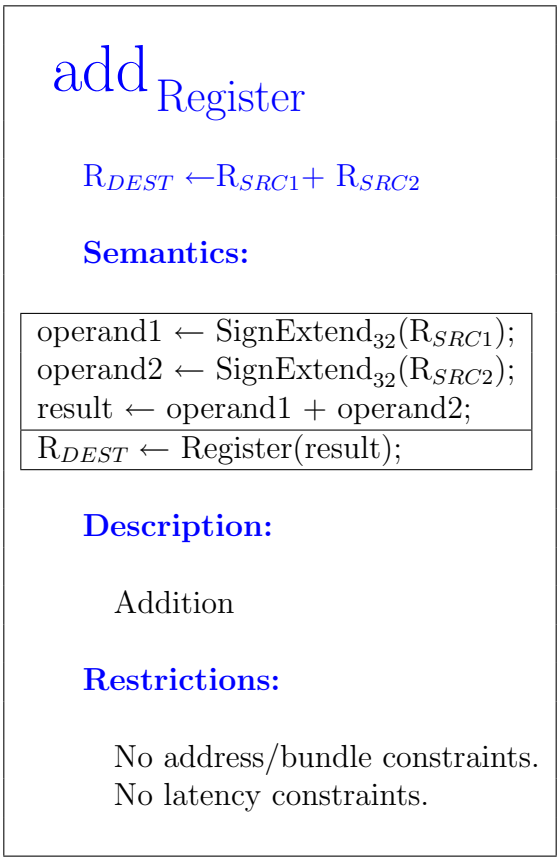
The instruction set includes 32-bit logic, arithmetic, and test instructions (see Table 2.1 for a list of all the instructions that have been used). Before providing the list of instructions, we present, first, how the instructions are encoded inside a bundle and second, an example to illustrate how an instruction is specified.

Bundle Encoding

An instruction bundle consists of between one and four consecutive 32-bit words, known as *syllables*. Each syllable encodes either an instruction or an extended immediate. The most significant bit of each syllable (bit 31) is a *stop bit* which is set to indicate that it is the last in the bundle. A syllable will therefore look like:

Instruction Specifications

Many instructions have an *immediate* form. The name of the instruction carries an optional subscript which is used to distinguish between instructions with different operand types, for example, register and immediate format integer instructions. An example of an instruction is given below followed by its descriptions [88].



The subscript Register indicates that both source operands are registers. The semantics specifies that the instruction is executed in two stages. In the first stage, which does not affect the architectural state of the machine, the values of the R_{SRC1} and R_{SRC2} registers are read, interpreted as signed 32-bit values and are assigned to temporary operands called operand1 and operand2. The statement

$$\text{result} \leftarrow \text{operand1} + \text{operand2};$$

adds the two integers operand1 and operand2, and assigns the result to a temporary integer called result. In the second stage, the statement

$$R_{DEST} \leftarrow \text{Register(result)};$$

gets executed if no exceptions have been thrown in the bundle, and updates the architectural state. The function Register converts the integer result back to a bit-field, discarding any redundant higher bits. This value is then assigned to the R_{DEST} register. There are no bundle constraints for this instruction, which means that up to four of these instructions can be used in a bundle; and there are no latency constraints either, which means that R_{DEST} will be ready for use by instructions in the next bundle. Finally, this instruction can not generate any exceptions. Instructions that are executed in the integer unit have no bundle and/or latency constraints while the one which

gets executed in the multiply unit have constraints such as "the instruction must be followed by 2 bundles before R_{DEST} can be read".

Branches are recognized as a major problem to exploiting parallelism. So a conditional select instruction `slct` helps improving the performance by eliminating branches and branch penalties, and at the same time reduces code size.

An important feature of the ST200 processors is the ability to shift an operand left by a fixed amount and add to another operand in one cycle. This is called *shift-and-add* instruction. For example,

`mov` \$R3 = 4 //R3 = 4
`sh4add` \$R0 = R1, R2 //R0 = (R1 << R3) + R2

In the above example R1 << R3 means that R1 is shifted left by an amount R3 $\in \{1, 2, 3, 4\}$. This type of instruction is very useful in computer arithmetic as it eliminates the need for separate shifts and can save code space, used register space (if the shifted operand value is reused), and can improve the overall performance.

Instructions

We give a precise account of the most useful instructions in Table 2.1. They are listed alphabetically for ease of use, followed by their definition and a short description. In short description *RD* is a destination register and Op_1 and Op_2 are two input operands. Most of the instructions listed have a latency of one cycle. All multiplication instructions take 3 cycles to execute on their respective processors. For example, `mul32` will take three cycles on ST231 but when it has to be performed on ST220, four $16 \times 32 \rightarrow 32$ multiplications emulate it which takes at six cycles.

2.3 Compiler

2.3.1 ST200 Compiler Overview

The purpose of the `st200cc` compilation driver is to translate a program written in the C language, into the ST220/ST231 assembly language so that it is suitable for assembly, linking, and execution. The assembler file is compiled using `st200as` and linked using `st200ld` to provide an ST200 binary image. All these phases are hidden using the driver tool `st200cc`.

STMicroelectronics developed the `st200cc` compiler based on the Open64 technology [11] and the LAO code optimizer [24], in order to replace the Multiflow Trace Scheduling compiler [10] originally used on the Lx processor. The `st200cc` compiler was released in early 2003 and is currently actively developed to support new ST200 cores and advanced optimizations.

The `st200cc` compiler uses the GNU C language front-end, and implements state-of-the art compiler optimizations. The `st200cc` compiler is closely compatible with the GNU C compiler, both at the driver level, and on C language extensions (GNU Compiler Collection project [5]). The processor-independent compiler optimizations available in the `st200cc` compiler are mostly inherited from the Open64 compiler technology. Other compiler optimizations that are specific to the ST200 family of processors were developed by STMicroelectronics at the HPC/STS Compilation Expertise Center. These include:

- The exploitation of the ST200 `slct` (select) instruction.
- Aggressive instruction selection including mapping of the user boolean variables to the branch registers.

`add`	Add	$RD \leftarrow Op_1 + Op_2$
`addcg`	Add with carry and generate carry	$RD \leftarrow Op_1 + Op_2 + C_{in}$; $BD \leftarrow C_{out}$
`and`	Bitwise AND	$RD \leftarrow Op_1 \wedge Op_2$
`andc`	Complement and bitwise AND	$RD \leftarrow (\tilde{Op_1}) \wedge Op_2$
`andl`	Test logical AND	$RD \leftarrow (Op_1 \neq 0)$ AND $(Op_2 \neq 0)$
`clz`	Count leading zeroes	$RD \leftarrow$ count of leading zeroes in Op_1
`cmpeq`	Test for equality	$RD \leftarrow Op_1 = Op_2$
`cmpge`	Signed compare equal or greater	$RD \leftarrow Op_1 \geq Op_2$
`cmple`	Signed compare equal or less	$RD \leftarrow Op_1 \leq Op_2$
`divs`	Nonrestoring divide stage	Integer division
`max`	Signed maximum	$RD \leftarrow Op_1$ if $Op_1 > Op_2$ else Op_2
`min`	Signed minimum	$RD \leftarrow Op_1$ if $Op_1 < Op_2$ else Op_2
`mulh`	$32 \times 16_H \rightarrow 32_L$ signed multiply	$RD \leftarrow Op_1 \times (Op_2 >> 16)$
`mulhhs`	$32 \times 16_H \rightarrow 32_H$ signed multiply	$RD \leftarrow (Op_1 \times (Op_2 >> 16)) >> 16$
`mulhs`	$32 \times 16_H \rightarrow 32_H$ unsigned multiply	$RD \leftarrow (Op_1 \times (Op_2 >> 16)) << 16$
`mull`	$32 \times 16 \rightarrow 32$ signed multiply	$RD \leftarrow Op_1 \times Op_2$
`mullh`	$16 \times 16_H \rightarrow 32$ signed multiply	$RD \leftarrow Op_1 \times (Op_2 >> 16)$
`mullhus`	$32 \times 16_L \rightarrow 16_L$ signed multiply	$RD \leftarrow (Op_1 \times Op_2) >> 32$
`mul32`†	$32 \times 32 \rightarrow 32_L$ signed multiply	$RD \leftarrow Op_1 \times Op_2$
`mul64hu`†	$32 \times 32 \rightarrow 32_H$ unsigned multiply	$RD \leftarrow (Op_1 \times Op_2) >> 32$
`nandl`	Logical NAND	$RD \leftarrow$ NOT$((Op_1 \neq 0)$ AND $(Op_2 \neq 0))$
`norl`	Logical NOR	$RD \leftarrow$ NOT$((Op_1 \neq 0)$ OR $(Op_2 \neq 0))$
`or`	Bitwise OR	$RD \leftarrow Op_1 \vee Op_2$
`orc`	Complement and bitwise OR	$RD \leftarrow (\tilde{Op_1}) \vee Op_2$
`orl`	Test logical OR	$RD \leftarrow (Op_1 \neq 0)$ OR $(Op_2 \neq 0)$
`shXadd`	Shift left `X` and accumulate	$RD \leftarrow (Op_1 << \texttt{X}) + Op_2$ where $\texttt{X} \in \{1, 2, 3, 4\}$
`shl`	Shift left	$RD \leftarrow 0$ if $amount > 31$ else $Op_1 << amount$
`shr`	Arithmetic shift right	$RD \leftarrow Op_1 >> distance$
`slct`	Conditional select	$RD \leftarrow Op_2$ if $Op_1 \neq 0$ else Op_3
`sub`	Subtract	$RD \leftarrow Op_1 - Op_2$
`xor`	Bitwise exclusive-or	$RD \leftarrow Op_1 \oplus Op_2$

†Specific to ST231 processor

Table 2.1: Main types of logical, arithmetic and test instructions available on the ST220 and the ST231 processors.

- Instruction scheduling.
- Software pipelining of the inner loops.
- Compiler intrinsics and builtins support.
- Reordering of object code to minimize instruction cache conflicts.
- Dead code and dead data elimination.
- Automatic generation of prefetch instructions.

- Profiling feedback optimization.
- Interprocedural analysis optimization.

The Open64 compiler has been re-targeted from the Intel IA64 to the STMicroelectronics ST200, except for the instruction predication, the global code motion (GCM), the instruction scheduler and the software pipeliner (SWP). Indeed, these components appeared to be too dependent on the IA64 architecture predicated execution model and on its support of modulo scheduling through rotating register files.

In the `st200cc` compiler, the Open64 GCM and SWP components are replaced by the ST200 LAO instruction scheduler/software pipeliner, which is activated at optimization level `-O3`. At optimization level `-O2` and below, only the Open64 basic block instruction scheduler is active.

The binary image can be executed on an ST220, or ST231 hardware target or on simulator.

2.3.2 Using `st200cc`

The `st200cc` compiler is similar to any command-line compiler. It is either invoked from a command line interpreter or from a Makefile, and it implicitly recognizes files by their extension. The C compiler is invoked using the `st200cc` command:

Example:
`st200cc` -S toto.c produces toto.s
`st200cc` -c toto.c produces toto.o
The final executable file does not need to have a specific file extension. If no output file name is specified through the `-o` option, the executable generated is named "a.out".

Example:
`st200cc` toto.c generates the executable a.out
`st200cc` toto.c -o toto.u generates the executable toto.u

The `-g` option instructs `st200cc` to generate symbolic information for debugging. The `-g` option may be used with optimizations up to level `-O2`. The options in Table 2.2 below control optimization levels.

Option	Description
`-O0`	No optimization
`-O1`	Minimal optimization
`-Os`	Optimization without code size expansion
`-O2`	Global optimization
`-O3`	Aggressive Optimization

Table 2.2: Optimization options.

Since `-O3` is not relevant in our library we use `-O2`. The level `-O3` enables more aggressive unrolling and software pipelining, but can incur code size increase.

2.3.3 Inlining Options for `st200cc`

The `-inline`, `-noinline` and `-INLINE` options are provided to control inlining of functions. Inline function expansion is performed for function calls that the compiler estimates to be frequently

executed. These estimations are based on a set of heuristics. The compiler might decide to replace the instructions of the call with code for the function itself (inline the call). These specific compiler options can be used to gain maximum performance in applications. It avoids the cost of a function call but the code size increases significantly depending on the number of functions inlined.

2.3.4 Intrinsic Functions

The `st200cc` compiler recognizes a number of intrinsic operators (e.g. `__lzcntw`) which can be used to produce assembly language statements that otherwise could not be expressed through standard ANSI C/C++. Access to integer and DSP arithmetic from C and C++ is supported by this extensive set of intrinsic functions. We uses some intrinsic functions, particularly to perform multiplications such as obtaining the most significant 32 bits of a 48-bit product produced by two 16 and 32-bit operands. This type of multiplication can not easily be performed using the simple multiplication operator ($\times$) and thus we depend on intrinsic functions like `__st220mulhhs`, `__muluw`, etc.

ST200 intrinsics have been modelled as C functions, which act as executable specifications. This has the benefit that models can be used to develop DSP algorithms on a workstation, that are immediately and safely ported to the ST220 and ST231. The functions defined in the ST200 Application Programming Interface provide access to low-level features, such as ST200 leading zero count (`lzc`) instruction and to medium level features, such as the International Telecommunication Union [7] (ITU) and European Telecommunications Standards Institute[3] [3] (ETSI) *basic operators*. All intrinsic functions are especially optimized by C and C++ compilers and are also delivered as standard C models for application validation on a workstation.

The intrinsic functions support has been fully developed in assembly language, validated using their C semantic models and Data Generation Language [2] and regenerated through an automatic flow as expansion function in the Open64 intermediate representation. The compiler either generates open code or emits call to library functions, leading to very efficient 64-bit and DSP arithmetic. The low-level integer arithmetic support directly uses the intrinsic functions. All the compiler-generated arithmetic intrinsics are held in the library `libgcc.a`.

2.4 Toolset Overview

The ST200 Micro Toolset is a set of tools that allow C and C++ programs compiled for a ST200 target to be simulated on a host workstation or executed on a ST200 target. The ST200 development system is a *cross-development environment*, that is, the target machine for which the code is developed (for example, ST220) is distinct from the host machine on which the development tools run. As there are many situations in which the availability of a target is not possible, a *software simulator* is provided on which to run and evaluate code. This has the advantage of allowing software to be developed in parallel with the detailed design of the hardware, as well as permitting software considerations to exert a reciprocal effect on hardware design (in cases where the hardware architecture has not been finalized).

[3]Many of the standard algorithms in Wireless and Wireline communications such as GSM (Global System for Mobile Telecommunications) or AMR-NB for UMTS (Adaptive Multi Rate Narrow Band for UMTS) encoders, are provided by the ITU and ETSI, as ANSI C code using 16-bit fractional arithmetic.

To specify the fractional arithmetic model which is foreign to ANSI C, a set of subroutines are used implementing basic fractional operations. As these basic fractional operations are provided as ANSI C source codes, an algorithm complying with the ITU or ETSI compiles on any ANSI C compiler.

The ST200 Micro Toolset is mainly intended for embedded application developers. It includes the whole set of tools that manipulate ST200 object files, including the ST200 assembler, compiler, linker, load/run tool, debugger and archiver. Here, ST200 assembler files are translated into ST200 object files that the linker merges to produce an ST200 executable image. This image file does not run natively on the host workstation and requires an interpreter or real hardware to be executed. The current version of the ST200 Micro Toolset that supports the ST220 and ST231 cores is R4.1 (June 2005). All the results and performances of FLIP have been measured using the R4.1 toolset and similarly for the validation and testing of all the operations. Figure 2.4 shows the main components of the ST200 Micro Toolset.

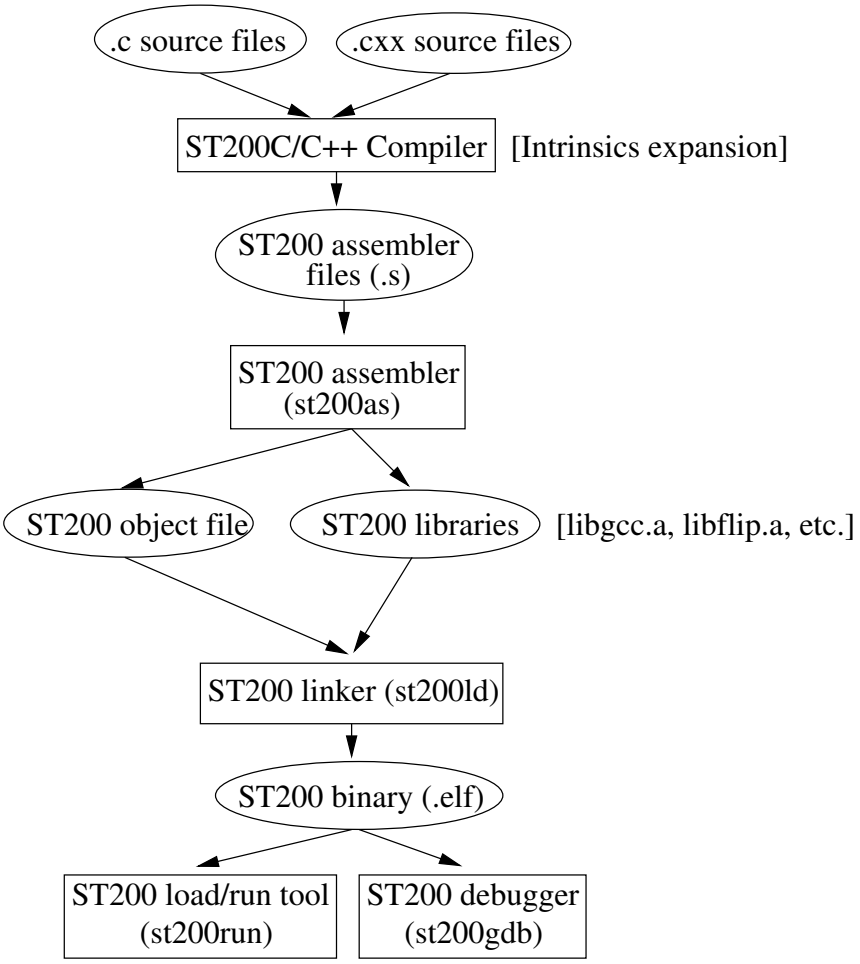


Figure 2.4: Components of the ST200 Micro Toolset interface (Courtesy: figure taken from [90] and has been modified).

The st200 debugger when launched with GUI provides a "ST2x0 Statistics" window which displays the current simulator statistics to the screen as shown in Figure 2.5. In the statistics window, two quantities are interesting for us. They are *Num completed bundles* and *Num completed instructions* which are the number of bundles/instructions that were executed during the program execution. The performance is measured in *number of cycles* regardless of any cache effects.

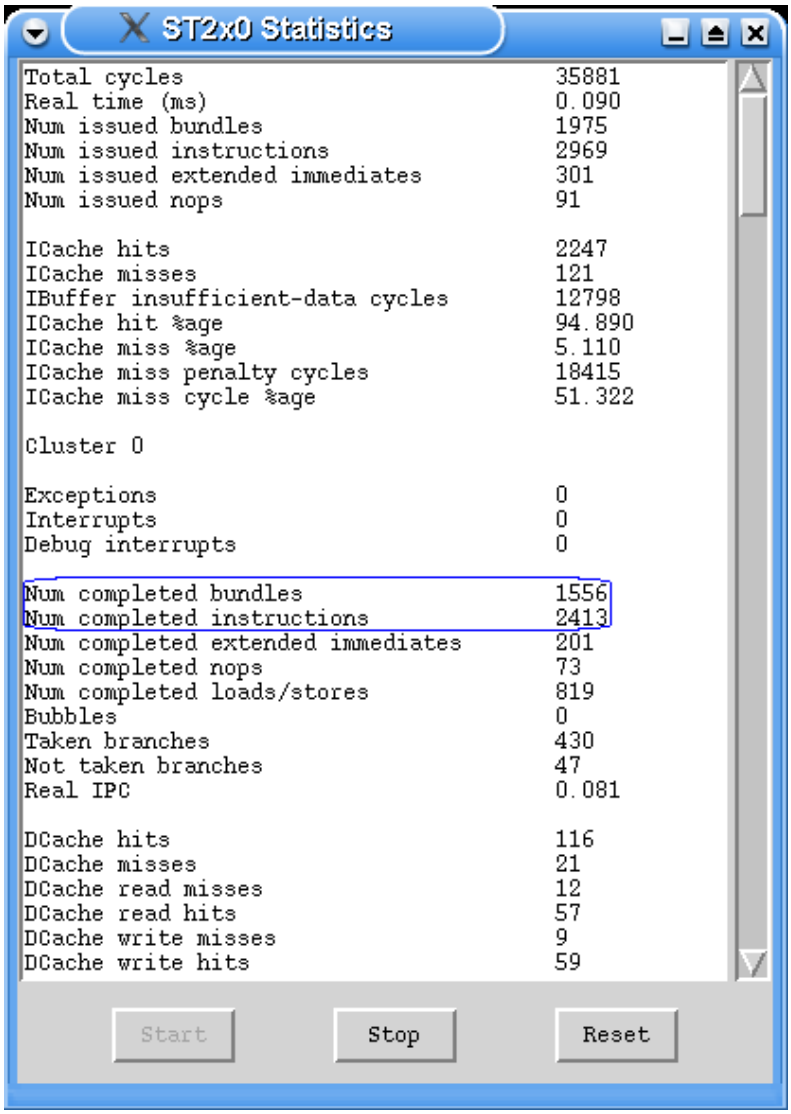


Figure 2.5: The ST2x0 Statistics window.

Chapter 3

FLIP

FLIP, a Floating-point Library for Integer Processors has been developed to support floating-point arithmetic on the processors ST220 and ST231 from STMicroelectronics. These processors are capable of performing computations with integers only. So FLIP acts as a software layer which emulates floating-point operations by performing integer arithmetic internally. To date, FLIP provides many different floating-point operations in the single precision format and for few operations it is almost compatible with the IEEE standard. The initial motivation for FLIP was an already existing library for basic operations optimized by STMicroelectronics And now, due to better optimization techniques employed in FLIP, it is faster than the original library for each operation supported and for some operations, a three or four fold increase in performance has been obtained.
This chapter presents this library, its overview, its purpose and, its interaction with the user/application, the compiler and the processor. The different options to build this library and the dependencies between different C source and header files constituting FLIP are presented in the section "Organization of FLIP". All the operations supported have been classified into three different versions of FLIP. This chapter concludes by a fruitful discussion about the achievements of FLIP.

3.1 The FLIP Library

FLIP, a Floating-point Library for Integer Processors is a C library for the software support of IEEE single precision floating-point arithmetic for processors without floating-point hardware units such as VLIW or DSP processor cores for embedded applications. In particular it is targeted for two different processors, ST220 and ST231 from STMicroelectronics. This library provides several levels of compliance with the IEEE standard; depending on the choice to support subnormal numbers and/or all the rounding modes. This type of compliance allows us to estimate the cost of supporting each option. Knowing the cost beforehand helps to fulfill the specific requirements of different applications. The flags (see Section 1.6) are not supported. Also, we do not stick to the rules of returning a silent or quiet NaN as defined by the standard. The library has been optimized for STMicroelectronics processors of the ST200 family. The optimizations done show a significant improvement in performance in comparison to the original, already optimized, library. The testing

ensures the correctness of the result produced by FLIP. This library is distributed under the Lesser General Public License (LGPL).

3.2 How to Use the FLIP library?

This section helps to understand the following points:

- How a user or an application can use FLIP to perform floating-point operations on the integer processors ST220/ST231?
- How the ST200 compiler interacts with FLIP?
- How floating-point emulation is done?
- How FLIP interacts with the target processors?

The Figure 3.1 illustrates how the different interfaces work together in order to use the library.

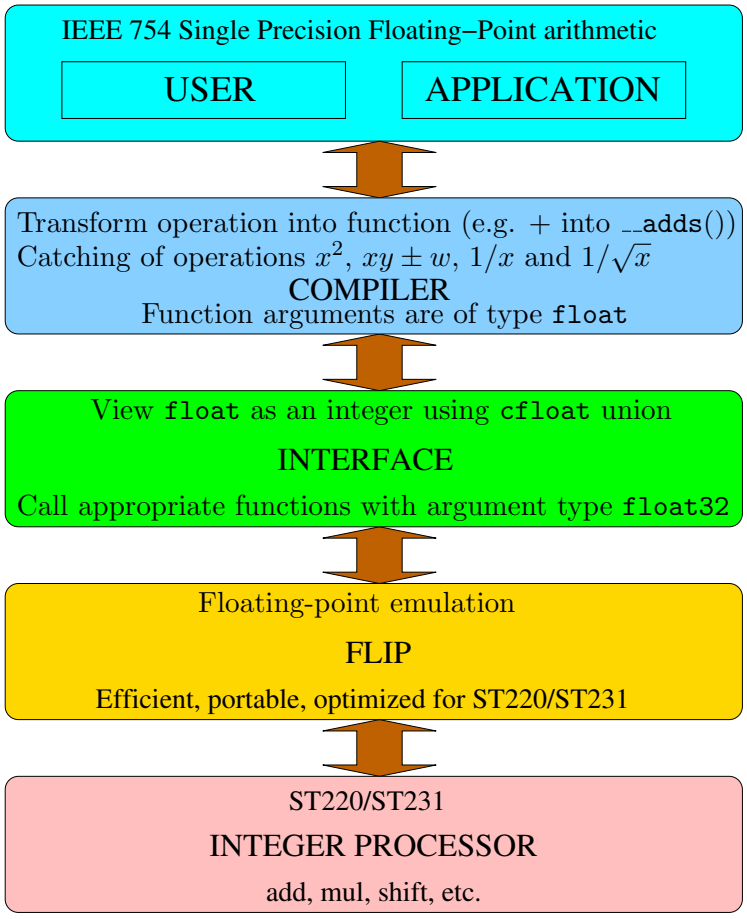


Figure 3.1:

3.2.1 User/Application Interface

At the user or application level, the statement to perform floating-point operations, say an addition, can be written as

```
float x,y,z;
z=x+y;
```

where `x,y,z` are single precision floating-point numbers.

3.2.2 Compiler Interaction with FLIP

After the above statements are issued at the user/application level, they are transformed by the compiler, by linking with the library of intrinsic functions, to

```
float x,y,z;
z=__adds(x,y);
```

where `__adds` is an intrinsic function to add two operands and "s" in `__adds` denotes that these operands are single precision floating-point numbers. In Figure 3.1, for clarity, the compiler, the library FLIP and the interface between them are shown separately but the complete library has been integrated within the ST200 compiler. This does not mean that the library is not portable, in fact it can be ported for any compiler with proper links, as explained in Section 3.3. The entry points of the FLIP library have been defined in an internal interface through which a `float` bit pattern can be viewed as a `float32` value. This is done by defining a

```
union cfloat
```

as

```
typedef union
{
 float32 f32;
 float f;
 unsigned int i;
} cfloat;
```

where `float32` is defined as

```
typedef float32 unsigned int;
```

The above `union cfloat` allows to view a 32-bit word as either a `float32` value or a standard `float` value or an unsigned integer value. The `float` arguments of `__adds` are viewed as arguments of type `float32` and the appropriate function for addition in FLIP is called as shown below.

```
float __adds( float x, float y )
{
 cfloat _x, _y, _r;
 _x.f = x;
 _y.f = y;
 _r.f32 =  __float32_flip_add_float32_float32( _x.f32, _y.f32 );
 return _r.f;
}
```

The nomenclature for the prototype of all the functions is as follows:

1. The prototype starts with `__float32` which is a return type for the function.
2. It is followed by `_flip_`OP where OP is the name of the operation.
3. In the end it is followed by as many `_float32` as the number of arguments a particular operation has.

For operations with one argument the prototype will be

```
__float32_flip_OP_float32( float32 );
```

where OP ∈ {`sqrt, sqr, reciprocal, invsqrt`}.
For operations with two arguments the prototype will be

```
__float32_flip_OP_float32_float32( float32 , float32 );
```

where OP ∈ {`add, sub, mul, div`}.
For operations with three arguments the prototype will be

```
__float32_flip_OP_float32_float32_float32( float32 , float32 , float32 );
```

where OP ∈ {`fma, fms`}.

In the above prototypes `sqr` is x^2, `invsqrt` is $1/\sqrt{x}$ and `fma, fms` is fused multiply and add/subtract, that is, $xy \pm z$. The floating-point operations are emulated by FLIP as described in Section 3.4. Inside FLIP only integer operations are performed as supported by the instruction set of the processors ST220/ST231.

3.2.3 Specific Compiler Optimizations

The optimizations at compiler level were done by STMicroelectronics. These optimizations aim at exploiting the non-trivial operations of FLIP such as x^2, $xy \pm z$, $1/x$ and $1/\sqrt{x}$. Before these optimizations, when x^2 was computed the multiplication operation was called with both the input operands as x. Now, the compiler can catch these operations and is able to call specific function. These optimizations further improved the latency of individual operations and run-time of some applications performing these operations (see Section 4.5).

3.3 Organization of FLIP

Figure 3.2 clearly depicts the organization of FLIP. It gives dependency of each C source and header file required to build and use the library FLIP, libflip.a. The library is completely independent and portable. All the intrinsic functions used have been inlined. In order to use the already built library one must link with "-lflip". The library can be built with different configuration options as follows:

For target processor:

- ST231: supported by default.
- ST220: to support use the option `--enable-st220`.

For subnormal numbers:

- not supported by default.
- to support use the option `--enable-subnormals`.

For rounding modes:

- round-to-nearest-even: supported by default.
- all IEEE rounding modes: to support use the option `--enable-all-rnd-modes`.

For example, when the library is built with `./configure` all the default options are selected. In order to build the library for the target processor ST220 with the support for subnormal numbers and all the rounding modes one must configure with `./configure --enable-st220 --enable-subnormals` All the different configuration options are not supported for all the arithmetic operations supported in FLIP. For example, when the low-performance implementation of division and square root is used, all the options are supported but when the high-performance implementation is used, only the default option is supported. This is due to the fact that when the library is built with default options it is faster than when it is built with the support for other options (see Table 3.1). Thus when performance is measured in order to compute the speed-up factor (see Table 3.4) in comparison to the original library from STMicroelectronics FLIP is built with default options.

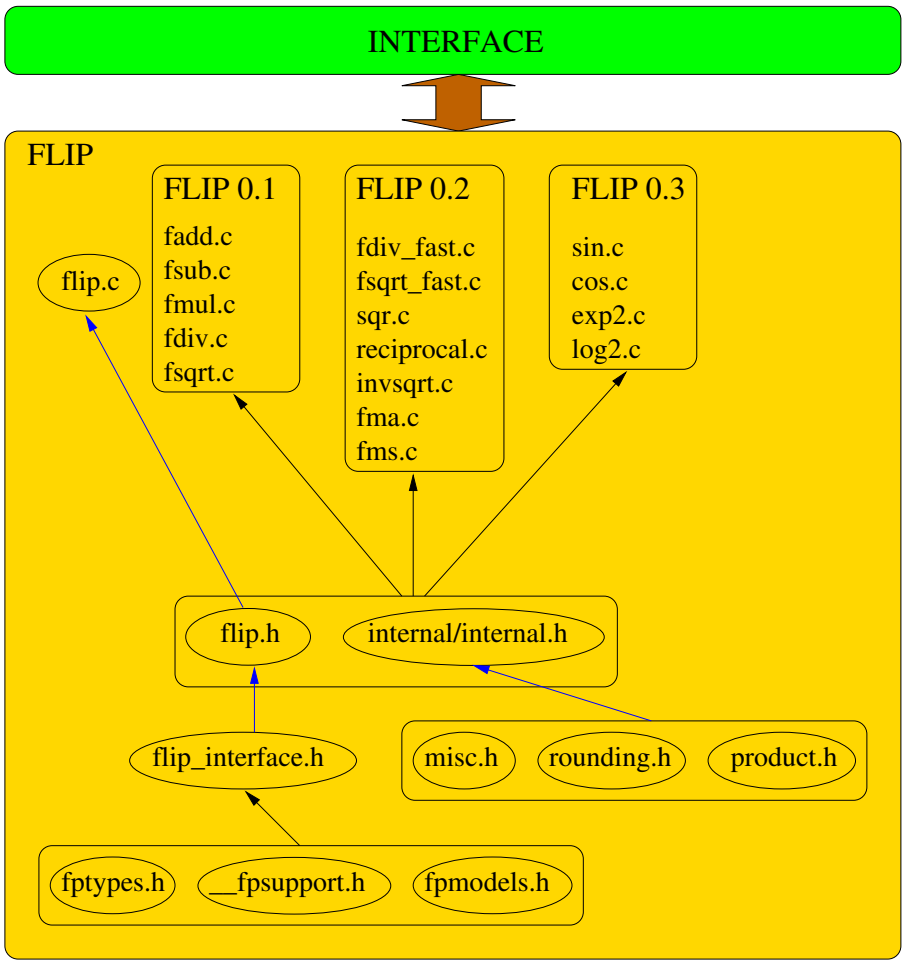


Figure 3.2: Organization of FLIP.

In Figure 3.2 we can see that all the arithmetic operations have been classified into three different versions whose features are explained in Sections 3.3.2, 3.3.3, 3.3.4. Before presenting the current three versions of FLIP we provide some details which are common to each operation implemented in FLIP, that is, information about the support provided by the header files in Figure 3.2 and some user level functions defined in "flip.c".

3.3.1 Details Common to the Different Versions

flip.c

The C source file "flip.c" contains the code of user level definition for setting the active rounding mode. It can be done by calling the function `flip_set_rnd_mode` and passing any one of the following four arguments.

1. `FLIP_RN` for round-to-nearest-even.
2. `FLIP_RU` for round towards positive infinity.
3. `FLIP_RD` for round towards negative infinity.
4. `FLIP_RZ` for round towards zero.

flip.h

It is the main header file of FLIP. This is the only file required to compile programs based on FLIP. It contains the following information:

- Definition of the following values (shown in hex representation which is an IEEE coding for a particular value):
  1. `FLIP_NaN` = 0x7FFFFFFF for NaN (one of the possible codings).
  2. `FLIP_ZERO` = 0x00000000 for $+0$.
  3. `FLIP_MZERO` = 0x80000000 for -0.
  4. `FLIP_INF` = 0x7F800000 for $+\infty$.
  5. `FLIP_MINF` = 0xFF800000 for $-\infty$.
- Declaration of functions for arithmetic operations (see Section 3.2.2 for prototype of all such functions).
- Definition of global variables:
  1. `flip_rnd_mode` $\in$ {`FLIP_RN, FLIP_RU, FLIP_RD, FLIP_RZ`}.
  2. `flip_status` used to record the exceptions occurred.
- Definition of `union cfloat` in the header file "flip_interface.h" which also includes the following three header files from the STMicroelectronics library.
  1. "fptypes.h": provides type definitions which are based on the STMicroelectronics library. Definitions which are used in FLIP are
     - `typedef unsigned bits32;`
     - `typedef int sbits32;`
     - `typedef unsigned float32;`

internal.h

This file contains all the internal FLIP definitions. Those definitions are only required to build FLIP; they are not accessible at the user level. In particular, internal.h contains

- The definition of some macros:
  1. `_FLIP_MAX`(x,y) returns the maximum of x or y.
  2. `_FLIP_MIN`(x,y) returns the minimum of x or y.
- The definition of some constants:
  1. `_FLIP_EXP_SIZE` = 8.
  2. `_FLIP_MANT_SIZE` = 23.
  3. `_FLIP_BIAS` = 127.
- The definition of some intermediate functions in the following three header files:
  1. "misc.h" contains FLIP internal miscellaneous definitions of functions:
     - `_flip_detect_exp_update`: detects post-normalization in advance.
     - `_flip_overflow_join`: detects overflow and join the exponent and mantissa part together. If overflow occurs then the exponent is set to 255 and the mantissa to 0, otherwise the original exponent and mantissa are joined together and returned.
     - `_flip_underflow`: detects underflow and returns either 0 or the smallest normalized number.
     - `_flip_declare_nan`: returns `FLIP_NaN`, whenever there is an invalid operation.
     - `_flip_declare_infinity`: returns either `FLIP_INF` or `FLIP_MINF`.
     - `_flip_declare_zero`: returns either `FLIP_ZERO` or `FLIP_MZERO`.
  2. "product.h" contains FLIP internal definitions of the following functions for product computations:
     - `_flip_mul_mant_with_sticky`: returns the product of mantissas with the computation of sticky bit from the discarded part.
     - `_flip_32h_umul_32_32`: returns the most significant 32 bits of a 64-bit product of two unsigned 32-bit values (alias is `__mulhuw` for ST200 compiler).
     - `_flip_32h_smul_32_16`: returns the most significant 32 bits of a 48-bit product of two signed 32-bit and 16-bit values (alias is `__st220mulhhs` for ST200 compiler).
     - `_flip_64_umul_32_32`: returns the 64-bit product of two unsigned 32-bit values (alias is `__mulun` for ST200 compiler).
  3. "rounding.h" contains FLIP internal definitions for rounding: `_flip_round` returns the 24-bit mantissa rounded according to the active rounding mode.

3.3.2 FLIP 0.1

This first version of FLIP provides the basic arithmetic operations $+, -, \times, /, \sqrt{}$. The input to these operations is a single precision IEEE floating-point operand which may be a normalized number, a subnormal number or a special value. The output can be obtained for any of the four rounding modes. As said before, appropriate configuration options must be provided when

building the library to support these features. Besides the default configuration option, that is, for the target processor ST231, without subnormal numbers and round-to-nearest-even, there exist three different combinations of the options to build FLIP 0.1.

1. Subnormals support and round-to-nearest-even.
2. No subnormals and all the four rounding modes support.
3. Subnormals and all the four rounding modes support.

These combinations allow us to estimate the cost of supporting each feature. For no subnormals support, if they occur in input they are considered as zero and thus treated in the same way as the special value zero. For support of all rounding modes, the user can set the active rounding mode dynamically, that is, by calling the internal function `flip_set_rnd_mode` and defining the global variable `flip_rnd_mode`. We say FLIP 0.1 is compatible with the IEEE standard when it supports both, subnormals and all the rounding modes.

In this version of FLIP, low-performance implementation of division and square root is used. This implementation is based on the classical *nonrestoring algorithm*[33, p.38] (also see Section 9.3.3). Multiplication operation involves a multiplication of the mantissas which can be performed using both kinds of multipliers available, either $16 \times 32 \rightarrow 32$ on ST220 or $32 \times 32 \rightarrow 32$ on ST231.

3.3.3 FLIP 0.2

This second version of FLIP implements the operations x^2, $xy \pm z$, $/, 1/x, \sqrt{x}$ and $1/\sqrt{x}$. Among these operations, x^2 and $xy \pm z$ are derived operations. It means that these operations are derivatives based on the basic operations and is faster than its originator. For example, operation $x \times y$ when $x = y$ is faster if computed as x^2 or y^2. Similarly, $xy \pm z$ can be computed faster than the multiplication xy followed by the addition $xy \pm z$. For division and square root, fast implementations based on the high-radix SRT and Goldschmidt algorithms are used. The reciprocal operation is a derivative of division. The square root operation is derived from the inverse square root as $\sqrt{x} = 1/\sqrt{x} \times x$.

The input(s) to these operations is a normalized number or a special value and the output can only be obtained for round-to-nearest-even mode. Thus, in order to build FLIP 0.2, default configuration options must be chosen (see Section 3.3). Since the implementations are optimized for both ST220 or ST231 and the processor ST231 is supported by default, the option to select ST220 must be enabled if necessary.

In this version, the implementation of all the operations involves the "trivial computation step" which is performed in the beginning. It filters the input operands which are zero and allows to reduce the computation time [76, 97]. It is a special treatment in the case where x, y, or z is zero.

3.3.4 FLIP 0.3

This third version of FLIP provides the following elementary functions:

- $\sin(x)$, $\cos(x)$ where the input operand x is a single precision floating-point number that lies in the reduced interval $[-\pi, \pi]$,
- 2^x and $\log_2 x$ where x is a single precision floating-point number.

Subnormal numbers and all rounding modes are not supported, so default configuration options must be chosen to build FLIP 0.3. Implementation of all the four functions involves the "trivial computation step" as in FLIP 0.2.

3.4 Floating-Point Emulation

3.4.1 Original Library

The previous floating-point support is an already optimized library from STMicroelectronics. This library is our reference benchmark to which we compare the performance of FLIP. It is a derived work from John Hauser's SoftFloat package [47] which is a general software implementation of all the operations in the IEEE standard, for both single and double precision floating-point numbers. STMicroelectronics has optimized the implementation of the 32-bit version (single precision) for the processors of the ST200 family (particularly ST220)with some relaxed characteristics: there is only one rounding mode (round-to-nearest-even) and support for subnormal numbers have been removed. It also uses specific intrinsic functions such as leading zero count to gain speed. See Section 3.5 for the performance of this library for the processors ST220 and ST231 and eventually for a comparison with FLIP.

3.4.2 FLIP

In this section we discuss about the strategy we have used for emulating floating-point operations. In all the C source files for arithmetic operations, the unpacking and packing of operands is the same. It is done by using extraction functions defined in "__fpsupport.h". After extraction, computation is done with the extracted exponent and mantissa. How these computations are performed in an optimized way is explained in Chapters 5 to 16 for each operation. Thus, each chapter for each operation deals with everything after the unpacking and before the packing stages, that is, each chapter deals with the core algorithmic part of the computation.

3.5 Performances

In this section, we provide all the performances concerning FLIP and we compare them with the original library optimized by STMicroelectronics which was from John Hauser's Softfloat package. In the end, we provide the speed-up factor for each operation. A speed-up factor of at least one has been achieved for all the operations. This approves of the better optimizations employed in FLIP.

Performances in number of cycles (equivalent to accounting for bundles and branch penalties and regardless of any cache effects) have been measured for all the operations, without support for subnormal numbers and only for round-to-nearest, both on ST220 and ST231 (Table 3.1). Note that the original library also does not provide support for subnormals and all the rounding modes. These figures are obtained when FLIP has been built with default options and these are the best timings FLIP has achieved. For each operation it is better than the original library. One should remark that for original library the timings for square and multiplication, and division and reciprocal are same. Also, the timings for inverse square root (306) and FMA (106) are sum of the timings combined for square root and division $(127 + 179)$, and for multiplication and addition $(45 + 61)$ respectively. This is because, the operations $(x^2, 1/x, 1/\sqrt{x}, FMA/FMS)$ are only provided in FLIP and thus the original library computes them thorough the basic operations. Moreover, the original library does not take advantage of the full-width multiplier available on ST231 whereas FLIP gains 2 cycles in case of ST231. The timings measured here for FLIP have been highly degraded because of the inclusion of the trivial computation step. If it is known beforehand that a particular application does not require this step then it is recommended to

built FLIP without this step. See the performance section in each chapter to get an idea of the improvement in timings when this step is not included.

	ST220		ST231	
	FLIP (default option) No subnormals Nearest even	Original library	FLIP (default option) No subnormals Nearest even	Original library
$+/-$	44/45	61/62	44/45	61/62
$\times$	38	45	36	45
$/$	62	179	51	177
$\sqrt{x}$	76	127	51	127
x^2	28	45	27	45
$1/x$	59	179	47	177
$1/\sqrt{x}$	96	306	67	304
FMA/FMS	60/61	106/107	59/60	106/107

Table 3.1: Performance of each operation as measured in number of cycles on ST220 and ST231

Now we provide the performances in number of cycles when FLIP is built with the options to support subnormal numbers and all rounding modes. The performances have been measured on ST220 (Table 3.2) and ST231 (Table 3.3). The foremost thing to observe here is that there is no difference in the timings between two processors except for the multiplication operation, in which case, a gain of 2 cycles has been obtained on ST231, as before. A common belief says that it is very costly to support subnormals and all the rounding modes. But we have shown in this research that contrarily to this belief, subnormals and all the rounding modes can be supported with an insignificant increase in latency and code size. The timings in Table 3.2 and Table 3.3 do not correspond to our point! This is due to the fact that the rounding modes are supported dynamically (user can change the rounding mode through a global variable) which entails some costly comparisons and as a consequence latency is increased. See the performance section in each chapter where the timings, when rounding modes are supported statically, are provided and these timings truly justify our point. Another important remark is that division and square root are implemented using the classical nonrestoring algorithms and thus they are slow in comparison to their timings in Table 3.1.

	ST220		
	Subnormals All modes	Subnormals Nearest even	No Subnormals All modes
$+/-$	74/75	44/45	67/68
$\times$	62	47	60
$/$	149	132	142
$\sqrt{x}$	129	123	125

Table 3.2: Performance of FLIP with support for subnormals and all rounding modes on ST220

Here we obtain the speed-up factor for FLIP in comparison to the original library. The speed-up factor for both libraries has been computed based on the timings provided in Table 3.1 for ST231. The speed-up factors show that FLIP is faster than the original library for each operation supported and for some operations, a three or four fold increase in performance has been obtained.

	ST231		
	Subnormals All modes	Subnormals Nearest even	No Subnormals All modes
$+/-$	74/75	44/45	67/68
$\times$	60	45	58
$/$	149	132	142
$\sqrt{x}$	129	123	125

Table 3.3: Performance of FLIP with support for subnormals and all rounding modes on ST231

Operations	$+/-$	$\times$	$/$	$\sqrt{x}$	x^2	FMA/FMS	$1/x$	$1/\sqrt{x}$
Speedup factor	1.38/1.37	1.25	3.47	2.49	1.66	1.79/1.78	3.76	4.53

Table 3.4: Comparison of FLIP (built with the default configuration options) with the library optimized by STMicroelectronics on the ST231 processor core.

Chapter 4

Validation and Application Results

The FLIP library has replaced the original floating-point library and has already been integrated in the industrial compiler for the ST200 family of processors. The library is used in embedded systems which have to be dependable. There might have dramatic economic consequences, if a STMicroelectronics customer produces thousands of faulty products based on this library. Because of all these reasons a high-level validation of FLIP which ensures its integrity and correctness of its results is indispensable.
This chapter discusses the procedures and protocols that have been followed for the validation of FLIP. In the first attempt, a two level validation is done. This involves validation of individual operations and then at higher level, validation through some numerical algorithms which performs a sequence of different operations. A complete validation, either of a particular range of input values or some particular operations, is done wherever possible. By complete validation we mean that there are some set of input values for which each operation can be tested and there are some operations which can be tested exhaustively (all the possible input values). After that, the discussion moves on to validation using the package TestFloat which tests if a floating-point implementation conforms to the IEEE Standard. The team of STMicroelectronics also helped in measuring performances of FLIP on their own developed applications. These are well known applications like the codec (coder/decoder) used in high-definition televisions and, a very famous computer game *Quake*.

4.1 Validation of FLIP: Definitions and Procedures

The validation of the FLIP library has been done using a battery of tests distributed on three different levels with the rigorousness and completeness of testing increasing at each higher level. These tests are designed keeping in view the two following points:

1. The extent up to which a particular operation can be tested (exhaustive or near exhaustive validation).

2. How complex (considering time and space) is it to test a particular operation due to the lack of hardware resources, fast testing algorithms, and/or availability of benchmarks.

The three levels of testing are:

Level 1:
Testing of an individual operation divided into two phases as described in Sections 4.2 and 4.3. At this level of testing an operation from the library FLIP "libflip.a" is fed with input value(s) and the output is compared with the result provided by the GNU C library "libgcc.a". GCC is *correctly rounded* for basic arithmetic operations but is said to be *faithfully rounded* for elementary functions (see Section 1.5 for these two types of roundings). For basic and additional operations, the output is said to be correct if it exactly the result provided by GCC up to the last bit. For elementary functions, we follow the *assumption* that the output is said to be correct if the difference from the GCC result is at most one ulp; the difference may be at most two ulps from the exact result.

Rather than testing all the values together, the range of input value is divided according to the type of value. In single precision format there exist 2^{32} (approximately 4.2 billions) different values consisting of 2 zeros $(+0, -0)$, 2 infinities $(+\infty, -\infty)$, $2^{24} - 1$ NaNs, $2^{32} - 2^{25}$ normalized values and $2^{24} - 2$ denormalized values. Special values are grouped together and tested for all the operations.

Level 2:
At this level of testing, we test sequences of operations instead of individual operations. This level is based on the run of some numerical algorithms. We have implemented four numerical algorithms as described in Section 4.4. These algorithms use intensively more than one operation on a fixed set of input values and produce a result which is known beforehand (for comparison). This confirms the correct implementation of the particular numerical algorithm as well as reinforces the fact that all the intermediate results produced by each operation used were correct. In short, the second level validates the overall numerical behavior of the library. The following algorithms have been implemented and successfully tested on our library.

Level 3:
This level of testing is based on the run of some applications as described in Section 4.5. These applications have been developed by HPC/STS CEC at STMicroelectronics which has validated and measured the performance of FLIP at this level.

4.2 Testing of Specific Values

4.2.1 Testing Fixed Input Values

In this category, individual operations are tested using long vectors of chosen patterns. A test pattern is composed of chosen argument(s) and the expected result of the operation. One says that the operation *passes* the vector test if for each pattern the computed result is the expected value. The patterns are chosen to test all the branches in the implementation as well as the behavior of the basic blocks of the implementation. We developed these vectors for each operation, each rounding mode with or without the support of subnormal numbers. The patterns include combinations of special values, some representable values, and values that lead to difficult rounding cases.

The vectors include exhaustive patterns for the special values. The representable values are some random values as well as the values such as $(N_{min}, N_{max}, N_{min}+1, N_{max}-1, (N_{max}-N_{min})/2$, etc.), which helps for testing a few border line cases for each operation. Figure 4.1 presents an example of difficult rounding case for multiplication and round-to-nearest-even mode. The product $x \times y$ with $x = 76$ and $y = 883013$ (x and y are exactly representable in single precision

format) should give a mathematical result of 67108988, but due to round-to-nearest-even mode, the returned value must be 67108992 for this format.

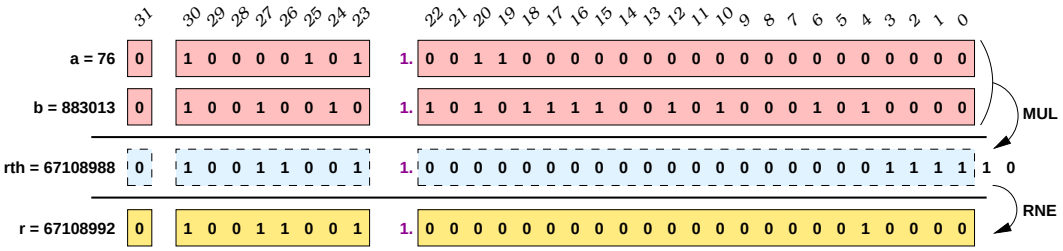


Figure 4.1: Validation example: round to nearest even of 76×883013.

4.2.2 Exhaustive Tests

The unary operations $\sqrt{x}, x^2, 1/x, 1/\sqrt{x}, 2^x, log_2(x)$ are tested exhaustively because in less than four hours all the input operands can be tested. It is possible since we are working in single precision. This means that apart from the special values all the other possible values have been tested for each operation for each rounding mode. Moreover, except for $\sqrt{x}$ all operations have been implemented to support normalized numbers and for the round-to-nearest-even mode only.

4.2.3 Testing with TestFloat

TestFloat [48] is a program for testing if a floating-point implementation conforms to the IEEE Standard for binary floating-point arithmetic. All standard operations supported by the system can be tested, except for conversions to and from decimal format. Any of the following machine formats can be tested: single precision, double precision, extended double precision, and/or quadruple precision.

TestFloat actually comes in two variants: one is a program for testing the machine's floating-point arithmetic, and the other is a program for testing the software implementation of floating-point arithmetic.

TestFloat checks the floating-point arithmetic by performing some simple tests. It compares the system's arithmetic with its own internal arithmetic implemented in software. Numerous test cases are generated for each operation and these test cases are based on simple pattern tests intermixed with weighted random inputs. Enough cases are generated so as to ensure almost the rounding of adds, subtracts, multiplies, and simple operations like conversions. These tests check all boundary cases concerning underflows, overflows, invalid operations, subnormal numbers, zeros, infinities, and NaNs. For operations like addition and multiplication, literally millions of test cases can be checked.

For each operation, TestFloat can test all the four rounding modes. TestFloat verifies the numeric results of an operation and also checks whether the exception flags are raised. TestFloat is not particular about the bit patterns of NaNs that appear as function results (sNaN = qNaN). Any NaN is considered as good a result as another. This laxness can be overridden so that TestFloat checks for particular bit patterns within NaN results.

4.3 Testing of Random Values

Testing exhaustively the operations with two input values, that is, testing approximately 2^{64} (18 trillion) different values is a very complex task and is nearly infeasible due to lack of resources and time.

To test such binary operations individually, random input values are chosen for both operands. The output for each random input is compared with the result provided by "libgcc.a" for that operation. Input values are values returned by the function "mrand48()" which generates pseudo-random numbers using the linear congruential algorithm and 48-bit integer arithmetic [9]. It generates a sequence of 48-bit integers according to the formula:

$$X_{n+1} = (aX_n + c) \mod m, \quad \text{where} \quad n \geq 0.$$

The parameter m is 2^{48}, hence 48-bit integer arithmetic is performed. The parameters a and c are given by: a = 0x5DEECE66D, c = 0xB. The value returned by mrand48() is a signed long integer uniformly distributed between -2^{31} and 2^{31}. The values a and c are called *seed* values. If the initial seed values are not changed then a same set of random numbers will be generated. Another function "srand(unsigned int)" is used which will seed the random number generator to prevent random numbers from being the same every time and to allow more pseudo-randomness. System time is used as an argument to srand as it is always changing.

An operation with three input values such as $xy \pm z$ (FMA/FMS) has been tested on the "Power PC G5" machine which has a hardware implementation of this operation. Testing all the combinations of special values for this operation is also done on this machine. Hardware implementation allows to conform that this operation gives indeed a correctly rounded result in spite of performing two floating-point operations in conjunction.

We conclude here that all the operations supported by FLIP have passed all kinds of tests performed at Level 1.

4.4 Testing through Numerical Algorithms

At this level of testing four algorithms have been implemented and successfully tested on the FLIP library. These algorithms test sequences of operations over a fixed set of input values and produce a result which is known beforehand.

Dot Product

Let vector A be $[1, 2, \ldots, n-1, n]$ and vector B be $[n, n-1, \ldots, 2, 1]$, their dot product is

$$DP = [1, 2, \ldots, n-1, n].[n, n-1, \ldots, 2, 1]^T.$$

In summation notation it can be written as

$$P = \sum_{i=1}^{n} i(n-i+1) = n^3/6 + n^2/2 + n/3$$

which allows to compute the theoretical value for a particular n. The default value of n used is 200 for which $DP = 1353400$.

Numerical Integration

Numerical integration is computation of the numerical value of a definite integral. Various methods exist for numerical integration. The one which we will use here is called numerical integration using the rectangles method. If $f(x)$ is the function that has to be integrated over some interval $[a, b]$ then let the interval be divided into n equal parts of length $h = \frac{b-a}{n}$. Now, divide the area under the curve $f(x)$ into n rectangles and sum up the area of all the rectangles. For validation we have evaluated numerically

$$NI = \int_1^2 x^{-1} dx.$$

The theoretical result is $NI = \ln 2 \approx 0.6931471$.

Order-4 Runge-Kutta Method

This method is used for numerical solving of differential systems. A major disadvantage of Taylor series method is the requirement of the evaluation of higher order derivatives. If the higher order derivatives of the function $f(x, y)$ are required then the Runge-Kutta method compensates this requirement by evaluating $f(x, y)$ at some points within the range x_0 to x_n. The fourth order formula is

$$\begin{aligned} k_1 &= hf(x_i, y_i) \\ k_2 &= hf(x_i + \tfrac{1}{2}h, y_i + \tfrac{1}{2}k_1) \\ k_3 &= hf(x_i + \tfrac{1}{2}h, y_i + \tfrac{1}{2}k_2) \\ k_4 &= hf(x_i + h, y_i + k_3) \\ y_{i+1} &= y_i + \tfrac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \end{aligned}$$

where $k_1, \ldots, k_4$ are h times different approximations to the slopes at n intermediate equidistant points denoted by x_i. The step is calculated as $h = (x_n - x_0)/n$. With the implemented Runge-Kutta method we have tried to solve numerically $y' = x/4 - y/4 + 2$ with the initial condition $x_0 = y_0 = 0$. Evaluation is done for $n = 100$ intermediate points, $x_n = 0.5$ being the last point. The already known solution is $y(x) = x + 4 - 4e^{-x/4}$.

Gaussian Elimination

Gaussian elimination is a method used for determining the solutions of a system of linear equations $Ax = b$, where

$$A = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \ldots & a_{nn} \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}, \quad \text{and} \quad x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}.$$

In this method, a sequence of elementary row operations are applied to an augmented matrix in a specific order and a reduced row echelon form is produced. The diagonal element of a matrix which divides the other elements is called the pivot. Partial pivoting is the interchanging of rows and full pivoting is the interchanging of both rows and columns. The method we have implemented solves the linear system through partial pivoting. We solve a $n \times n$ system for different values of n and for which the solution vector is $[1, 2, 3, \ldots, n]^T$. The system with $n = 4$ is given below.

$$A = \begin{bmatrix} 0 & 0 & 1 & 4 \\ 0 & 1 & 4 & 1 \\ 1 & 4 & 1 & 0 \\ 4 & 1 & 0 & 0 \end{bmatrix}, \quad b = \begin{bmatrix} 19 \\ 18 \\ 12 \\ 6 \end{bmatrix}.$$

It can easily be verified that the solution of this system is $[1, 2, 3, \ldots, n]^T$.

Apart from validating the overall numerical behavior of FLIP some performance measurements have also been done on these four numerical algorithms. These algorithms can be treated as light applications that use a sequence of different operations. The goal here is to study the cost of supporting different options in FLIP for complete applications. For each algorithm, we have used several values of the parameters (number of intermediate points for numerical integration and Runge-Kutta methods, size of vectors and matrices for dot product and Gauss elimination methods). In Figure 4.2, we report, for each algorithm, relative computation timings of FLIP with respect to the original library.

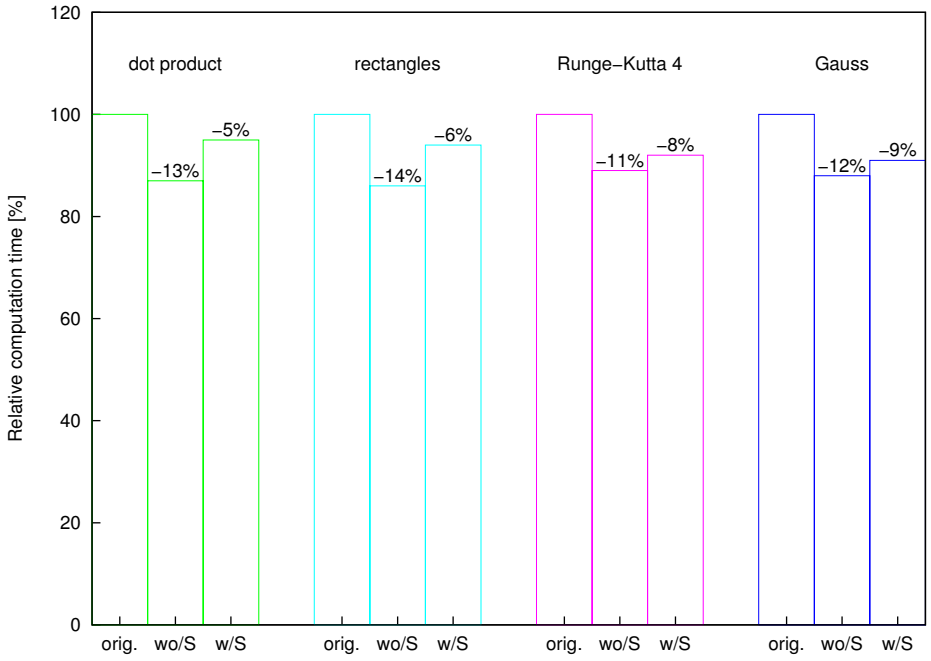


Figure 4.2: Comparison of relative computation timings for the various validation algorithms.

A $11-14\%$ speed improvement has been achieved for FLIP without the support of subnormal numbers (wo/S). When subnormal numbers are supported (w/S), the improvement is about $5-9\%$.

4.5 Applications: Testing and Performances

4.5.1 SBC: Sub-Band Codec Benchmark

The Bluetooth SBC is an audio coding/decoding system having a low-computational complexity and is designed for Bluetooth audio and video applications to obtain high quality audio at moderate bit rates. Bluetooth SBC is based on the low-complexity, low delay audio coder presented in [23]. SBC is mandatory for applications that implement the Bluetooth Advanced Audio Distribution Profile (A2DP). STMicroelectronics has optimized SBC for ST200 family of processors and has used it as a benchmark for evaluating performance of FLIP.

Performance Analysis

Performance of the SBC decoder was measured through three different libraries (floating-point supports) and the analysis gives an insight into relative speed-up factors and provides a statistics

on the type of operations, and on the trivial operand computation. The three libraries have been compared as shown in Figure 4.3. The figures on the arrows are the speed-up factors with the faster library being at the head of the arrow and the slower one at the tail. In order to be able to report the improvement in the performance after specific compiler optimizations (see Section 3.2.3) we use FLIP 0.2 and FLIP 0.2x (FLIP 0.2 extended). FLIP 0.2 supports the basic and additional operations *without* the catching of the operations such as x^2, $xy \pm z$, $1/x$ and $1/\sqrt{x}$. FLIP 0.2x supports all operations of FLIP 0.2 *with* the catching of these operations at the compiler level. Note that all the speed-up are computed on cycles without cache-effects.

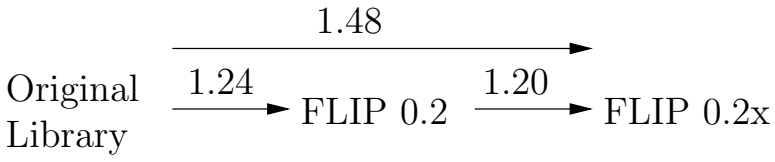


Figure 4.3: Relative speed-up factors for three different libraries for SBC decoder.

Finally, a speed-up factor of 1.48 for FLIP 0.2 in comparison to the original library has been reported. The inclusion of trivial computation step has really helped in improving the performances as can be observed in Table 4.1 and Table 4.2. Also, Table 4.2 shows that more than 87% of additions and multiplications performed are associated together to produce a multiply-and-add operation.

Function	nb. of calls	nb. of trivial operations	ratio
`__adds`	12800	3960	30.93%
`__subs`	1600	751	46.93%
`__muls`	12869	1943	15.09%
`__divs`	35	22	62.85%

Table 4.1: Statistics for trivial operands in SBC in case of FLIP 0.2.

Function	nb. of calls	nb. of trivial operations	ratio
`__adds`	1600	751	46.93%
`__subs`	1600	751	46.93%
`__muls`	1669	341	20.43%
`__divs`	35	22	62.85%
`__madds`	11200	3238	28.91%

Table 4.2: Statistics for trivial operands in SBC in case of FLIP 0.2x.

4.5.2 DTS-HD: Digital Theater Surround high-definition Benchmark

The same kind of performance analysis has been done for DTS-HD (Digital Theater Surround high-definition). The three libraries have been compared as shown in Figure 4.4 below.

Performances achieved after the inclusion of trivial computation step can be observed in Table 4.3 and Table 4.4. Table 4.4 shows that more than 75% of additions and multiplications performed are associated together to produce a multiply-and-add operation.

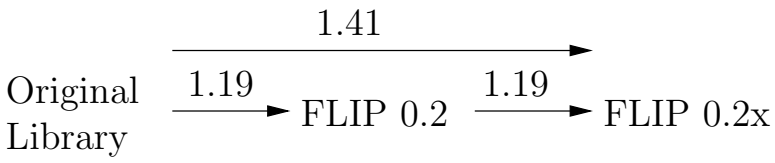


Figure 4.4: Relative speed-up factors for three different libraries for DTS-HD.

Function	nb. of calls	nb. of trivial operations	ratio
`__adds`	440417	107132	24.32%
`__subs`	61509	24999	40.64%
`__muls`	399465	35754	8.95%
`__divs`	16	3	18.75%

Table 4.3: Statistics for trivial operands in SBC in case of FLIP 0.2.

Function	nb. of calls	nb. of trivial operations	ratio
`__adds`	112692	75822	67.28%
`__subs`	61503	24996	40.64%
`__muls`	71722	22376	31.19%
`__divs`	13	0	62.85%
`__madds`	327725	31512	9.61%
`__msubs`	15520	2	0.012%
`__nmadds`	0	0	0.00%
`__nmsubs`	6	6	100.00%
`__recips`	3	0	0.00%
`__squares`	12	12	100.00%

Table 4.4: Statistics for trivial operands in SBC in case of FLIP 0.2x.

4.5.3 Quake

Quake is a well known compute-intensive computer game. FLIP 0.2x allowed to speed up the frame rate of the Quake game running on a ST231 Linux Board from 12.6 to 16.2 fps (frame per second), that is almost a 30% speed-up.

Part II

Addition, Multiplication and Related Operations

Chapter 5

Addition and Subtraction

Floating-point addition and subtraction are the most frequent floating-point operations in almost all kinds of applications. They are also often employed when performing more complex operations like multiplication and division. Conceptually, they are the simplest operations, returning either the sum or difference of two floating-point numbers. In practice, floating-point addition/subtraction can be slow due to its serial operations like alignment shift, normalization shift and rounding. Therefore, reducing the latency of this operation is vital to the high-performance floating-point support. Concerning hardware implementation much research has been done on improving its performance [17, 71, 73, 81, 82, 83, 38]. For software implementation, which is our case here, many hardware focussed optimizations do not follow. Improvements in software implementation is directly related to many different issues, for example, how aggressively the compiler can exploit the instruction-level parallelism or how efficiently it can schedule the instructions. Moreover, there are many algorithmic and architectural constraints to deal with. In this chapter we will first recall the basic notions and the basic algorithm of floating-point addition/subtraction. After that, we will present the implementation of each step of this basic algorithm accompanied with the optimization techniques employed to improve the overall latency of floating-point addition/subtraction. Our optimized implementation has helped to achieve a speed-up factor of 1.45 in comparison to the previous implementation in an already optimized library.

5.1 IEEE Floating-Point Addition/Subtraction

The inputs for the addition or subtraction are

1. Two floating-point operands x and y represented by (s_x, e_x, f_x) and (s_y, e_y, f_y) respectively.
2. One of the four rounding modes to specify how to round the exact result.

The output is a single precision floating-point number $z = \diamond(x \pm y)$, where $\diamond$ denotes the active rounding mode.

In addition to the fact that floating-point addition/subtraction is much more complex to implement than other basic operations, what intricates the situation more is that addition, may actually be a subtraction or vice-versa, depending on the signs of the operands.

Practically, it is not useful to implement two different independent operations, one for floating-point addition and another for floating-point subtraction. Any one of these operations can be implemented to perform both, by flipping the sign of the second operand. For example, if addition is implemented then the subtraction $x - y$ can be reduced to an addition by flipping the sign of y as $x + (-y)$.

Generally, only the floating-point addition is implemented to take care of the floating-point subtraction too. Here we focus only on the floating-point addition, that is, $x + y$ and/or $x + (-y)$. We denote the floating-point operation by fop, where $fop = 0$ indicates addition and $fop = 1$ indicates subtraction. The sign logic (see Table 5.1), depending on the fop and the signs of the operands, computes the effective operation (eop). It determines whether the *mantissas* of the floating-point operands have to added or subtracted. The sign logic also computes the sign of the result.

After the extraction of the sign, exponent, and fraction of each operand and reinsertion of the implicit bit one has the normalized mantissas m_x and m_y. We now present the basic description of the floating-point addition which gives a general idea of how it can be performed.

1. If any of the input operand is a special value, the result is known beforehand, thus all the computation can be bypassed.

2. Mantissas are added or subtracted depending on the effective operation:

$$\hat{S} = \begin{cases} m_x \pm (m_y \times 2^{e_y - e_x}) \times 2^{e_x} & \text{if } e_x \geq e_y, \\ ((m_x \times 2^{e_x - e_y}) \pm m_y) \times 2^{e_y} & \text{if } e_x < e_y. \end{cases}$$

   In order for the addition or subtraction to take place the binary point must be aligned. To achieve this, the mantissa of the smaller operand is multiplied by $2^{\text{difference of the exponents}}$ (see example 1). This process is called *alignment*.

3. The exponent of the result is the maximum of e_x and e_y; $e_z = \max(e_x, e_y)$.

4. If eop is addition, a carry-out can be generated and if eop is subtraction, cancellation might occur (see examples in Section 5.3). In each case normalization is required and consequently the exponent e_z is updated.

5. The exact result $\hat{S}$ is rounded to fit in the target precision. Sometimes rounding causes an overflow of the mantissa; a post-normalization is required.

6. Determine exceptions by verifying the exponent of the result.

Floating-point operation (fop)	Sign of the operands	Effective operation (eop)	Sign of the result (s_z)
ADD	same	add ($eop = 0$)	s_x
ADD	different	subtract ($eop = 1$)	s_x if $\lvert x \rvert \geq \lvert y \rvert$ else s_y

Table 5.1: Sign logic computes the effective operation and the sign of the result. In case of subtraction, it is reduced to addition by flipping the sign of the second operand.

Example 1: Let $x = -1.10011 \times 2^{-2}$ and $y = 1.10001 \times 2^{-5}$. In order to perform $x + y$, one needs a common exponent and so m_y must be aligned as shown below.

		2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}		
	m_x	1	. 1	0	0	1	1				$\times$	2^{-2}
$-$	m_y		. 0	0	1	1	0	0	0	1	$\times$	2^{-2}
		1	. 0	1	1	0	0	1	1	1	$\times$	2^{-2}

Once the alignment is done, the effective operation must be computed. Here, the sign logic says that $eop = 1$ and so $m_x - m_y$ is performed. What happens next is normalization, rounding and exception determination. How these steps are performed and the issues involved in their *implementation* are explained in Sections 5.3 and 5.4.

In example 1 above, since $|x| \geq |y|$, m_y is aligned and $m_x - m_y$ is performed but what if $|x| < |y|$? Obviously, m_x will be aligned and $m_y \pm m_x$ will be performed. But generally in hardware implementation, in order to have *only one alignment shifter*, mantissas are exchanged. This process is called *swapping*. Two pseudo-codes are compared below which shows the purpose of swapping.

```
read(x,y)
d = e_x − e_y
if abs(x) ≥ abs(y) then
    m_y = m_y >> abs(d)
    Ŝ = m_x ± m_y
else
    m_x = m_x >> abs(d)
    Ŝ = m_y ± m_x
```

```
read(x,y)
d = e_x − e_y
if abs(x) < abs(y) then
    m_t = m_x
    m_x = m_y
    m_y = m_t
m_y >> abs(d)
Ŝ = m_x ± m_y
```

In our software implementation too, we swap the mantissas as it solves the dilemma of either shifting m_x or m_y and then performing either $m_x \pm m_y$ or $m_y \pm m_x$. Once swapping is done, we are sure that it is m_y which has to be aligned always. Generally, the sign of d (sign of $m_x - m_y$ when $d = 0$) decides whether to swap the mantissas or not, and $|d|$ is the amount of alignment shift.

5.2 Special Values

A significant amount of computation can be bypassed when any of the input operand is $\pm\infty$, ± 0 or NaN. Instead of following the steps of the basic description, an already defined path is followed to handle special value cases. The minimum computation required on such input operands is the extraction of their sign, exponent and fraction. Once we have these fields, few comparisons are done to arrive at the final result. Table 5.2 summarizes the possible results of the floating-point addition of two numbers x and y.

Table 5.2 can be and is generally implemented by performing four comparisons with the values of e_x and e_y. The pseudo-code on the left in Figure 5.1 below verifies all the cases in the table by checking the condition $\{e_x, e_y\} \cap \{0, 255\} \neq \emptyset$, that is, whether the biased exponent of the input operands is either 0 or 255. The pseudo-code on the right in Figure 5.1 shows the methodology of handling special values in FLIP. This approach is advantageous as it allows to verify the existence of a special value through one comparison instead of four. Indeed, if the existence of a special value is confirmed, then a sequence of comparisons are performed to arrive at the result. The goal

$+$		y					
		NaN	$+\infty$	$-\infty$	$+0$	-0	y
x	NaN	NaN	NaN	NaN	NaN	NaN	NaN
	$+\infty$	NaN	$+\infty$	NaN	$+\infty$	$+\infty$	$+\infty$
	$-\infty$	NaN	NaN	$-\infty$	$-\infty$	$-\infty$	$-\infty$
	$+0$	NaN	$+\infty$	$-\infty$	$+0$	$+0$	y
	-0	NaN	$+\infty$	$-\infty$	$+0$	-0	y
	x	NaN	$+\infty$	$-\infty$	x	x	$\diamond(x+y)$

Table 5.2: All the possible predefined values of the sum $\diamond(x+y)$ when x or y is a special value.

behind this approach is that the special values are rare and so their handling should not influence the computation with normalized values. In the pseudo-code below, $e_{x_{spl}}$ and $e_{y_{spl}}$ are computed through some arithmetic and/or logic operations which are performed in *parallel* and, identifies the existence of a special value. Finally, the max instruction which takes one cycle, helps to compute the maximum of these two values and a single comparison with e_{spl} allows to confirm the existence of special values. More about this optimization technique can be found in Section 5.5.2.1.

if $e_x = 255$ **then**
  if $f_x \neq 0$ **then return** x;
  if $e_y = 255$ **and** $f_y \neq 0$ **then return** y;
  if $s_x = s_y$ **then return** x;
  return NaN; **if** $e_{s_y} = 0$ **then return** x;
  return $\pm\infty$;
if $e_y = 255$ **then**
  if $f_y \neq 0$ **then return** y;
  if $e_x = 0$ **then return** y;
  return $\pm\infty$;
if $e_x = 0$ **then return** y;
if $e_y = 0$ **then return** x;

$e_{spl} = \max(e_{x_{spl}}, e_{y_{spl}})$;
if $e_{spl} = 254$ **then**
if $d > 0$ **then**
  if $e_x = 255$ **then**
    if $f_x \neq 0$ **then return** x;
    if $f_y = 0$ **then return** x;
if $d < 0$ **then**
  if $e_y = 255$ **and** $f_y \neq 0$ **then return** y;
  if $e_y = 0$ **then return** y;
if $d = 0$ **then**
  if $e_x = 255$ **then**
    if $f_x \neq 0$ **then return** NaN;
    if $f_y \neq 0$ **then return** NaN;
    if $s_x = s_y$ **then return** x;
    return NaN

Figure 5.1: Basic pseudo-code on the left, and an optimized way on the right for handling special values.

5.3 Addition of Mantissas

In this section we discuss the implementation issues involved in adding the mantissas. It deals with different types of cases which are encountered while swapping, aligning and adding the mantissas.

We must recall that the floating-point addition operation supports the subnormal numbers and when an operand is a subnormal number ($e = 0$ and $f \neq 0$), then there is no hidden 1. Consequently, the exponent of such an operand is set to $e = 1$ and no implicit bit is added, so that $m = 0.f$.

Once the exponent difference $d = e_x - e_y$ is computed, mantissas are swapped if necessary, and m_y is prepared for alignment. The mantissa m_y is shifted right by $|d|$ bits. One way to obtain correctly rounded result is to compute $m_x \pm m_y$ exactly and then round the result. In single precision where exponent is represented using 8 bits, $|d|$ lies in the range $[0, 254]$, alignment is nearly impractical for higher values of $|d|$. So, to achieve this goal, computation is performed using three extra bits (G for guard bit, R for round bit and S for sticky bit). These bits are determined once the alignment is done. Figure 5.2 illustrates two possible cases for alignment. In both cases, when m_y is shifted right, instead of considering all the discarded bits, G and R are denoted as the bits at position 25 and 26 and, sticky bit S is computed by ORing all the bits which were shifted right past R. The least significant bit L at position 24, will be included in the final result. The bits G, R and S only facilitate the rounding process as explained later.

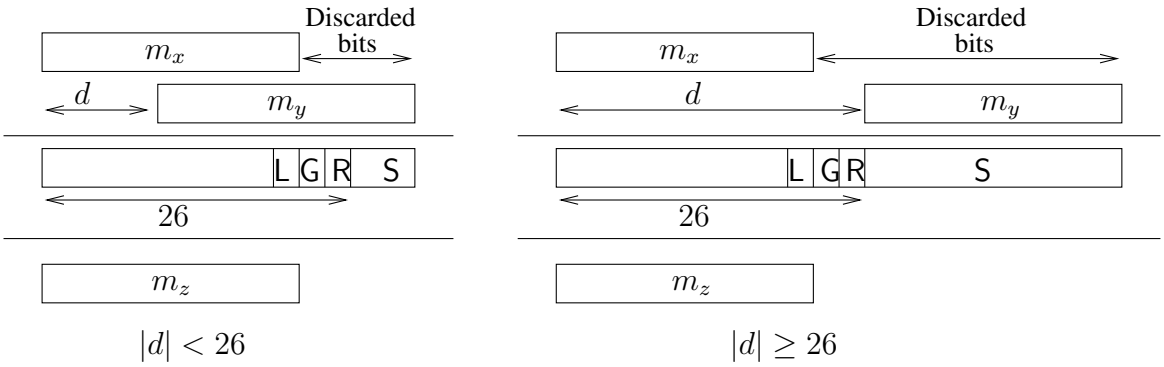


Figure 5.2: Illustration of the two cases while aligning the mantissa.

In implementation, to accommodate these bits, the mantissa is shifted left by three bits as shown in the pseudo-code below.

read(x, y)
$f_x =$ extract_frac(x)
$m_x = (f_x \quad | \quad$ 0x00800000$) << 3$

Figure 5.3 shows the way of storing G, R and S on a 32-bit register. It also shows that the maximum alignment shift is 26 bits. When m_y is shifted right by $d \geq 26$ bits, since the sticky bit S is computed by ORing all the discarded bits after position 26, two cases exist as

1. In case of normalized mantissa the implicit one goes at the position of the sticky bit.
2. In case of denormalized mantissa S $= 1$ since $y \neq 0$.

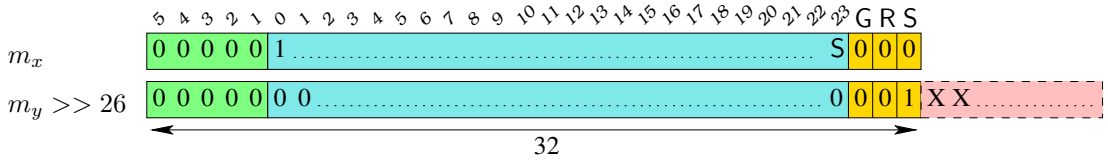


Figure 5.3: Illustration of the fact that alignment requires a shift of at most 26 bits.

In both cases, $d \geq 26$ would not change the contents of the register in which m_y is stored and so in this situation it suffices to set S $= 1$. Generally, only G and/or S $=$ R OR S are required

depending on the rounding mode and few specific cases. We now explain through some examples when these bits are required.

Example 2: Let $x = 1.011 \times 2^2$ and $y = 1.001 \times 2^{-22}$. After the alignment process mantissas are added with G, R and S bits as shown below.

						L	G	R	S		
m_x	1. 0	1	1	00000000000000000	0	0	0	0	0	$\times$	2^2
$+ \quad m_y$	0. 0	0	0	00000000000000000	0	0	1	0	1	$\times$	2^2
$\hat{S}$	1. 0	1	1	00000000000000000	0	0	1	0	1	$\times$	2^2
$\triangle(\hat{S})$	1. 0	1	1	00000000000000000	0	1				$\times$	2^2
$\nabla(\hat{S})$	1. 0	1	1	00000000000000000	0	0				$\times$	2^2
$m_z = \circ(\hat{S})$	1. 0	1	1	00000000000000000	0	1				$\times$	2^2

After the sum $\hat{S}$ is computed, rounding is performed. For round towards $+\infty$ ($\triangle(\hat{S})$) it is sufficient to know whether any of the bits G, R or S are nonzero. Since OR of these bits is 1, a roundup is necessary and thus m_z is chosen as the next floating-point number. For round towards $-\infty$ ($\nabla(\hat{S})$) the sum is chopped after the bit L. For round to nearest we need to know the guard bit G but this is not enough to decide whether to round up to the next floating-point number. We also need to know whether the sum lies at the exact middle of two consecutive floating-point numbers, if not, then on which side of the exact middle. This is determined by OR of R and S. In this case, since ROR S = 1, a roundup is necessary. So this example shows that for addition of two mantissas, at most two extra bits are required for all the rounding modes.

Example 3: Let us consider another example where the sum is not normalized: $x = 1.11 \times 2^2$ and $y = 1.0\ldots011 \times 2^1$. After the alignment process mantissas are added with G, R and S bits as shown below.

					L	G	R	S		
m_x	1. 1	1	00000000000000000	0	0	0	0	0	$\times$	2^2
$+ \quad m_y$	0. 1	0	00000000000000000	1	1	0	0	0	$\times$	2^2
$\hat{S}$	10. 0	1	00000000000000000	1	1	0	0	0	$\times$	2^2
$\hat{S}_{norm}$	1. 0	0	10000000000000000	0	1	1	0	0	$\times$	2^3
$m_z = \circ(\hat{S}_{norm})$	1. 0	0	10000000000000000	1	0				$\times$	2^3

Because of the carry generated on the left, the sum is normalized (note that the exponent is updated) and the bits G, R and S change. The bit L of $\hat{S}$ becomes the new guard bit. Again, at most two additional bits are required, after normalization, for correct rounding. For round towards $\pm\infty$, it suffices to know OR(L, G, R, S) of $\hat{S}$. For round to nearest, since L and G are 1, a roundup is necessary.

Example 4: Now, we consider subtraction in the following example. Let $x = 1.0 \times 2^0$ and $y = 1.0\ldots011 \times 2^{-24}$. The mantissa m_y is shifted 24 bits right and the subtraction is performed with the bits G, R and S as shown below.

			L	G	R	S		
m_x	1. 0	00000000000000000000	0	0	0	0	$\times$	2^0
$-\ m_y$	0. 0	00000000000000000000	0	1	0	1	$\times$	2^0
$\hat{S}$	0. 1	11111111111111111111	1	0	1	1	$\times$	2^0
$\hat{S}_{norm}$	1. 1	11111111111111111111	0	1	1	0	$\times$	2^0
$m_z = \circ(\hat{S}_{norm})$	1. 1	11111111111111111111	1				$\times$	2^0

Because the most significant bit is canceled, normalization is required. The guard bit of $\hat{S}$ is needed to fill-in the place of the least significant bit of $\hat{S}_{norm}$, and the round bit becomes the new guard bit. Since $\mathsf{G} = 1$ and $\mathrm{OR}(\mathsf{R}, \mathsf{S}) = 1$, the final sum must be rounded up to $1.11\ldots1$.

The last example shows a specific case that if subtraction causes the most significant bit to cancel, then all the three extra bits are needed whatsoever is the rounding mode. Note that, if two most significant bits are canceled, then x and y are so close that $x - y$ will be exact and no extra bits are needed at all.

We have provided three examples which show that correctly rounded result can be obtained by using three extra bits. In our implementation, we maintain these three bits with a 24-bit mantissa as shown in Figure 5.3. As can be seen in these examples that for rounding only two bits are required, so we recompute, once normalization is done, two bits and call them as guard and sticky bits.

5.4 Rounding

In this section we deal with different cases when normalization is required and finally provide the rounding function for each rounding mode. If $1 \leq \hat{S} < 2$, no normalization is required and thus $\hat{S} = \hat{S}_{norm}$. If it is not the case, two possibilities occur:

1. In case of effective addition, since $1 \leq m_x, m_y < 2$, $\hat{S} \in [1, 4)$. Normalization is done as $\hat{S}_{norm} = \hat{S} >> 1$ and the exponent is incremented as $e_z = e_z + 1$.

2. In case of effective subtraction, cancellation might occur and $\hat{S}$ may have leading zeros. In this case, leading one is predicted by computing the number of leading zeros in $\hat{S}$. Normalization can be done as $\hat{S}_{norm} = \hat{S} >> shl$ where shl is computed as $shl = \mathtt{lzc}(\hat{S})$. The function stands for "leading-zero-count". The exponent is decremented as $e_z = e_z - shl$.

We now consider the use of bits G, R and S to perform rounding in different modes. After normalization if we compute OR of R and S bits and call it as the new sticky bit $\mathsf{S} = \mathsf{R}$ OR S then only two additional bits are required. Now we have a 26-bit $\hat{S}_{norm}$ represented by a binary string $\mathrm{S}_{norm}[0:25]$ and we denote $\mathrm{S}_{norm}[23] = \mathsf{L}$, $\mathrm{S}_{norm}[24] = \mathsf{G}$ and $\mathrm{S}_{norm}[25] = \mathsf{S}$. Rounding requires an addition of a bit $rnd \in \{0, 1\}$ to L and if it causes an overflow, it is necessary to normalize again by incrementing the exponent. The Table 5.3 below shows the computation of rnd. Here, $'$ indicates bit-negation.

In many implementations (including ours), round to nearest is performed by

Ronding mode	rnd
$\triangle(\hat{S}_{norm})$	$s_z{}'$ AND (G OR S)
$\nabla(\hat{S}_{norm})$	s_z AND (G OR S)
$\mathcal{Z}(\hat{S}_{norm})$	0
$\circ(\hat{S}_{norm})$	G AND (L OR S)

Table 5.3: The rounding function for floating-point addition/subtraction.

1. always adding 1 to G which produces the rounding to nearest, and
2. correct the bit in the position L if there is a tie, that is, make $\mathsf{L} = 0$ if G AND $\mathsf{S}' = 1$.

After rounding is performed the mantissa of the result is obtained as $m_z = \diamond(\hat{S}_{norm}) >> 2$. Sometimes rounding produces m_z as $10.0\ldots00$; post-normalization is required. Since the floating-point representation uses the hidden bit, only the exponent is updated. Once the result mantissa is obtained, the result exponent e_z is verified for overflow/underflow, in which case a pre-defined result is returned. Finally, the fraction f_z, the exponent e_z and the sign s_z are packed as a 32-bit floating-point number (z) which is returned as result.

5.5 Optimizations

In this section we first present the standard optimization techniques employed in the implementation of floating-point addition. Next, we give arguments as to why we can not benefit from these techniques in our implementation. Followed by this, we present the optimization techniques that helped us to improve our software implementation in comparison to the one implemented in the original library. The two implementations are compared in Figure 5.4. The implementation in the original library has already been optimized once by STMicroelectronics. We use this implementation as a reference and try to optimize it again.

5.5.1 Standard Optimization Techniques

In hardware, optimizations are mainly focussed on parallelizing and reordering some computation steps. This parallelism comes at the cost of additional hardware as in the case of *dual path* optimization from [38]. This technique exploits some inherent characteristics of floating-point addition/subtraction like the alignment shift and the normalization shift can be made mutually exclusive. When the effective operation is subtraction and $d \in \{0, 1\}$, a few significant bits of the result might become zero. In this case, there is a need for a left normalization shift. The dual path technique separates this particular case into a path with a dedicated left shifter while the cases for effective addition or effective subtraction with $d \notin \{0, 1\}$ pass through the regular path. The special path is called the cancellation path (where the leading bits are possibly cancelled) or the *CLOSE* path (where the exponents are close to each other) while the regular path is called the *FAR* path. Generally, this implementation is pipelined so that a new operation can begin each cycle, increasing the throughput. Another kind of dual path technique based on a nonstandard separation that considers not only the magnitude of the exponent difference but also whether the operation is effective subtraction is given in [84]. This kind of separation maintains the advantages of the standard dual path technique, as described above, and in addition, separates the rounding to take place only in one path.

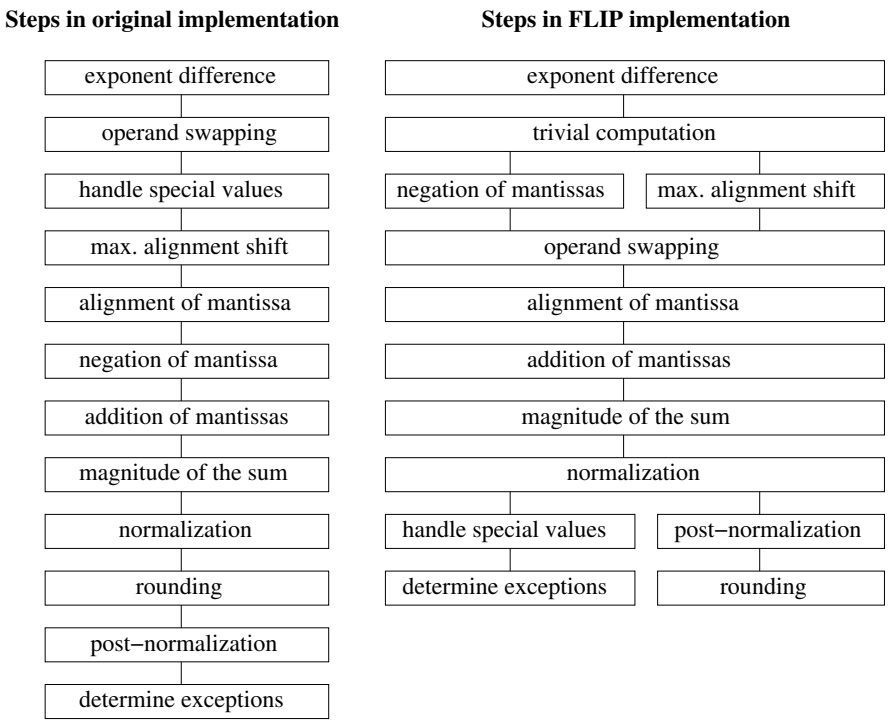


Figure 5.4: High level comparison of the implemented floating-point addition, original Vs FLIP.

In software implementation, compiler plays an important role. Several computation steps can be optimized so that compiler can efficiently schedule the instructions in order to increase the *bundle fill rate* (see Section 2.4) for the 4 instructions/bundle target VLIW processor. Our fundamental strategy in optimizing any implementation is

1. To diminish the number of empty bundles (`nop`) (Section 2.4)
2. To fill up the bundle with the maximum number of possible instructions and
3. To reduce the total number of bundles required (control code size).

The dual path technique is not worth adopting in our case chiefly due to the following reasons:

- Several jumps due to comparisons: in addition to the branch created due to the comparison that ought to be done for trapping the special values, extra comparisons leading to a branch must be done, two if FAR path is eventually chosen and three in case of CLOSE path. Compiler is unable to schedule such comparisons with other computations and is forced to take such jumps introducing some `nop` bundles.
- Code size: In hardware, since FAR path may require full-length alignment shift but never needs more than 1 bit left normalization shift (*vice versa* for CLOSE path), it may be interesting not to perform 1 bit shift explicitly. But in software it is better to perform both the full-length and the 1 bit shift explicitly on both the paths, otherwise advance prediction of 1 bit shift may cost another comparison. Moreover, both such shifts have same latency on target processors. Also, addition of the mantissas must be done on both paths. So this unavoidable duplication of instructions leads to an increased code size.
- Effective operation: extra computation (or comparison) is required in order to know whether effective addition or effective subtraction is to be performed.

- Leading-one-prediction: In CLOSE path, in case of subtraction yielding cancellation, it is possible to predict the number of leading zeroes in the result directly from the input operands instead of waiting for the completion of the mantissa addition. This prediction will require extra computation and will definitely cost a latency of more than 1 cycle. We can use the available `lzc` instruction on our target processors whose latency is 1 cycle.

5.5.2 Optimization Techniques in FLIP

Now we present the optimizations done in our implementation. The driving idea behind optimizations is to perform minimum possible comparisons (tests) which may lead to a branch. Such comparisons are costly (latency of 3 cycles), particularly when no other computation can be done in parallel. This happens, due to compiler limitations, when comparisons are required to be done in the beginning since other computations are dependent on the result of this comparison. This situation creates more or less a sequential execution of a fragment of the code. We have tried to separate out the steps such as handling of special values and post-normalization from the main execution path of the code. These steps can be executed independently, in parallel, and do not influence directly the execution path.

5.5.2.1 Handling of Special values

It is certain that the implementation consisting of four consecutive comparisons to handle special values (pseudo-code on the right in Figure 5.1) will offset much of the performance gained by the optimizations of other steps. If both the operands are normal values, which is the case almost every time, four tests must be always performed on the main execution path before computing the final result. The solution that we provide to sideline such comparisons uses the fact that the occurrence of special values in input is extremely rare and thus the computation with normal values should not suffer.

Note that this solution does not reduce the number of comparisons if any of the operand is indeed a special value, in which case we follow more or less the implementation with four comparisons. The goal here is to reduce the number of comparisons from the critical path and filter all the special values by performing only one comparison after a "max" computation [18] (see the pseudo-code on the right in Figure 5.1).

The handling of special values uses the following fact. If $\{e_x, e_y\} \in \{0, 255\}$ which gives $\{\hat{e}_x, \hat{e}_y\} \in \{-1, 254\}$ where $\hat{e}_x = e_x - 1$ and $\hat{e}_y = e_y - 1$ then the internal representation of -1 and 254 using binary strings can be given as

$$-1[31:0] = [11\ldots11111111]$$

and

$$254[31:0] = [00\ldots011111110]$$

respectively. In the range $\{-1, 254\}$, -1 and 254 are the only integers such that their masking with 0x000000FE produces 254. Let us represent $\hat{e}_x$ and $\hat{e}_y$ by the binary strings $\hat{E}_x$ and $\hat{E}_y$ as

$$\hat{e}_x = \sum_{i=0}^{31} \hat{E}_x[i]2^i \quad \text{and} \quad \hat{e}_y = \sum_{i=0}^{31} \hat{E}_y[i]2^i.$$

After computing

$$e_{x_{spl}} = \sum_{i=1}^{7} \hat{E}_x[i]2^i \quad \text{and} \quad e_{y_{spl}} = \sum_{i=1}^{7} \hat{E}_y[i]2^i,$$

we see that condition $\{e_x, e_y\} \in \{0, 255\}$ is equivalent to condition $\max(e_{x_{spl}}, e_{y_{spl}}) = 254$. This solution performs some precomputation with the exponents e_x and e_y which is always done in parallel with other computations. With this solution, it suffices to perform one comparison after a max computation to *check* the existence of special values. Once the existence is confirmed, four comparisons follow. This solution completely boycotts the handling of special values from the main execution path. It significantly improves the latency when the input operands are normal values (see Section 5.6 for a performance comparison).

One might think, when the original and our implementations are compared (Figure 5.4), about the difference in order in which the special values are handled. In our implementation we try to delay the handling of special values because of the following reasons:

- Again, the compiler is unable to schedule the comparison (after max operation) with other computations if special values are to be checked in the beginning. By delaying, the compiler is able to do this comparison in parallel to other computations and only in case of an operand being a special value, it is forced to take a branch.

- As said earlier, since special values are rare and we do not want to lower the performance for normal values, we try to do the computations even if the operands are special values and in parallel compute the max and perform the comparison and later change the results if required.

The disadvantage of this optimization technique is that the latency of the floating-point addition increases in case of special values. There is a gain of 55% in latency for the original library from STMicroelectronics in comparison to FLIP. It is due to the fact that in the original library, for any floating-point operation, checking for special values is the foremost step and thus the result is returned as quickly as special values have been detected. In FLIP, floating-point operation is performed first and the result is computed *before* this detection. If the input operands are detected as special values then the result is changed accordingly and is finally returned. Hence, this optimization slows donw the case of special values in FLIP.

5.5.2.2 Trivial Computation Step

When the library is built without any support of subnormal numbers and for round-to-nearest-even mode, the handling of special values has been modified by the inclusion of the "trivial computation step". The motivation behind this modification is application dependent. STMicroelectronics has found that the frequency of zero operands in input values was significantly very high in target applications (see Section 4.5 for details). So it was interesting to deal with zero operands quickly. This step, now, filters out all zero operands through explicit comparisons. These comparisons result in a branch which increases the latency of floating-point addition for normal values. But, the inclusion of this step can be justified by the significant gain in performance achieved for this application.

5.6 Performances

Performances in Section 3.5 give general timings for addition/subtraction with different types of support. In this section we will discuss how these timings are obtained and for which input values. In addition to this, the goal here is to study the impact of the following points on the general timings.

1. Influence of the trivial computation step.

2. Different input values:
    - Special values.
    - Normalized values.
    - Subnormal values.
    - Specific cases such as $-2 + 2$ for which the result is 0 in all rounding modes except round toward negative infinity, in which case it is -0.
3. Input values which cause overflow/underflow (no subnormal support).
4. Input values which cause gradual underflow (subnormal support).
5. Basic way of handling the special values.
6. Delaying the handling of special values.
7. Advancing the post-normalization step.

We have classified all the above cases into five sections and provide the different input values used and the timings obtained for each case. The general timings from Section 3.5 for addition/subtraction are given in the Table 5.4 below. For simplicity reasons, when FLIP is built with default support option, that is, with no subnormals and round-to-nearest-even, we call it FLIP with support A. Similarly, support B signifies subnormals with round-to-nearest-even, support C signifies no subnormals with all rounding modes and support D signifies subnormals with all rounding modes.

	A	B	C	D
ST231	44/45	44/45	67/68	77/78
ST220	44/45	44/45	67/68	77/78

Table 5.4: General timings for floating-point addition/subtraction on ST220 and ST231.

Since the routine for subtraction performs one XOR operation (1 cycle) and calls the routine for addition, the total number of cycles is one more than the addition. Also the timings for both ST220 and ST231 are always the same. We now give the details of each support option.

Support A: The input values are x = 0x40E00000 = 7 and y = 0x41300000 = 11. Trivial computation step is included and thus x and y are checked in the beginning for zero operands. See Section 5.6.1 for the influence of this step on the overall implementation.

Support B: The input values are x = 0x40E00000 = 7 and y = 0x41300000 = 11. There is no trivial computation step and the gradual underflow is not possible ($x + y = 18$). See Section 5.6.3 for the worst case timing for this support. One might say from the above timings that the cost of supporting subnormal is zero, which is not the case. To allow a fair comparison of support A and B we must suppress the trivial computation step so that the real cost can be obtained as given in Section 5.6.1.

Support C: The input values are x = 0x40E00000 = 7 and y = 0x41300000 = 11. Trivial computation step is included and its behavior is same as the support A except that other rounding modes

can be chosen dynamically (see Section 3.2). The comparisons done during the post-normalization and rounding steps account for the increased timings. See Section 5.6.2 for exceptional cases that might happen in the case when chosen rounding mode is negative infinity.

Support D: The input values are x = 0x40E00000 = 7 and y = 0x41300000 = 11. It comn=bines the features of support B and C except that there is no trivial computation step. The comparisons account for the increased timings. See Section 5.6.1 to know the real cost of supporting each feature.

5.6.1 Influence of the Trivial Computation Step

As said before this feature is supported only in support A and C, thus the general timings of support B and D are not affected. The general timings in Table 5.4 do not provide a fair comparison of all the supports since the true cost of supporting "subnormal numbers" and "all rounding modes" can not be judged because of the inclusion of trivial computation step. The timings below shows the impact of this step on normalized input values.

x, y	A	B	C	D
$7, 11$ (without)	36	44	60	77
$7, 11$ (with)	44	44	67	77

5.6.2 Different Input Values

Special values

$e_x, e_y \neq 255$	A	B	C	D
$0, y$	18	55	21	73
$x, 0$	18	49	21	67
$0, 0$	18	56	21	65

x, y	A	B	C	D
∞/NaN, y	48	49	49	67
x, ∞/NaN	54	55	55	73
∞/NaN, ∞/NaN	53	56	56	73

Subnormal values

x, y	A	B	C	D
$2^{-39}, 2^{-36}$	18	44	21	77

Specific case: $-2 + 2 = -0$ when rounding toward negative infinity.

x, y	A	B	C	D
$11, -11$	44	44	52	77
$2^{-39}, -2^{-39}$	18	44	21	57

5.6.3 Input Values Causing Overflow/Underflow

x, y	A	B	C	D
(overflow)	44	44	67	77
(underflow)	44	44	67	79

5.6.4 Handling Special Values

Basic way: pseudo-code on the left in Figure 5.1 is implemented by performing four comparisons. Since the handling is delayed, it is performed in parallel.

x, y	A	B	C	D
$7, 11$	42	47	71	86

Optimized way: pseudo-code on the right in Figure 5.1 is implemented by performing one comparisons after a max operation. Since the handling is done in the beginning, it slows down the general case (normalized input values).

x, y	A	B	C	D
$7, 11$	45	52	62	77

5.6.5 Post-Normalization Step

Generally, post-normalization is required once rounding is done but in our optimized implementation we perform this step before rounding by detecting the bit pattern of the mantissa and updating the exponent. Thus this step is made completely independent of the rounding step. The timings below show the impact of advancing this step.

x, y	A	B	C	D
$4, -2^{-26}$ (before rounding)	44	44	67	77
$4, -2^{-26}$ (after rounding)	49	50	64	76

Chapter 6

Multiplication

Multiplication can be considered as the next most frequent floating-point operation. The critical part in floating-point multiplication is the multiplication of the mantissas. This involves generation of partial products and their summation. Reducing the number of partial products or accelerating their summation is vital to the overall performance of floating-point multiplication. The basis of many optimization techniques that were described for floating-point addition apply equally to floating-point multiplication and other floating-point operations. In this chapter we explore two different methods of multiplying the mantissas. Both methods have been adapted to the rectangular multiplier of ST220 but can be modified in order to be used with the square multiplier of ST231. The latency involved in both methods is almost the same but we will see that one method is preferable to the other due to its algorithmic simplicity and higher adaptability to the target processors. Our implementation has achieved a speed-up factor of 1.29 compared to the floating-point multiplication supported in the original library.

6.1 IEEE Floating-Point Multiplication

The inputs for multiplication are:

1. Two floating-point operands x and y represented by (s_x, e_x, f_x) and (s_y, e_y, f_y) respectively.
2. One of the four rounding modes to specify how to round the exact result.

The output is a single precision floating-point number $z = \diamond(x \times y)$, where $\diamond$ denotes the active rounding mode.

After the extraction of the sign, exponent, and fraction of each operand and reinsertion of the implicit bit one has the normalized mantissas m_x and m_y. We now present the basic description of floating-point multiplication.

1. Verification of special values: if any of the input operand is a special value, the result is known beforehand and thus no computation is required.
2. Mantissas are multiplied to compute the product as

$$\hat{P} \leftarrow m_x \times m_y.$$

3. The exponent of the result is computed as

$$e_z \leftarrow e_x + e_y - 127.$$

In biased representation, the bias 127 (for single precision) has to be subtracted.

4. The sign of the result is computed as

$$s_z = s_x \text{ XOR } s_y.$$

5. The product might be unnormalized. In this case, normalization is required and consequently the exponent e_z is updated.

6. The exact result $\hat{P}$ is rounded according to the specified mode to produe the mantissa of the result m_z. In case of post-normalization, it is necessary increment the result exponent as $e_z \leftarrow e_z + 1$.

7. Determine exceptions by verifying the exponent of the result. When overflow occurs, the result is $\pm\infty$ and when underflow occurs, the result is zero or a subnormal number (if the gradual underflow is supported in the implementation).

The basic description provided a brief overview of the different steps involved in floating-point multiplication. We explain these steps and discuss their implementation issues in Sections 6.2, 6.3 and 6.4.

6.2 Special Values

In this section we deal with the situation when any of the input operand is a special value such as $\pm\infty$, ± 0 or NaN. The result of the operation is known in advance and thus no computation is required. Once the three components (sign, exponent and fraction) of input operands are obtained, few comparisons as illustrated by the basic pseudo-code on the left in Figure 6.1 are done to arrive at the final result. Table 6.1 summarizes the possible results of the floating-point multiplication of two numbers x and y.

$\times$		y					
		NaN	$+\infty$	$-\infty$	$+0$	-0	y
x	NaN	NaN	NaN	NaN	NaN	NaN	NaN
	$+\infty$	NaN	$+\infty$	$-\infty$	NaN	NaN	$\pm\infty$
	$-\infty$	NaN	$-\infty$	$+\infty$	NaN	NaN	$\pm\infty$
	$+0$	NaN	NaN	NaN	$+0$	-0	± 0
	-0	NaN	NaN	NaN	-0	$+0$	± 0
	x	NaN	$\pm\infty$	$\pm\infty$	± 0	± 0	$\diamond(x \times y)$

Table 6.1: All the possible predefined values of the product $\diamond(x \times y)$ when x or y is a special value.

Table 6.1 is generally implemented by performing four comparisons with the values of e_x and e_y as shown by the pseudo-code on the left in Figure 6.1. The pseudo-code on the right is an optimized way of handling the special values in FLIP. This approach is same as described in Section 5.2 for floating-point addition. See Section 6.5.1 to know more about this optimization technique.

if $e_x = 255$ **then**
  if $f_x \neq 0$ **then return** x;
  if $e_y = 255$ **and** $f_y \neq 0$ **then return** y;
  if $e_y = 0$ **then return** NaN;
  return $\pm\infty$;
if $e_y = 255$ **then**
  if $f_y \neq 0$ **then return** y;
  if $e_x = 0$ **then return** NaN;
  return $\pm\infty$;
if $e_x = 0$ **then return** ± 0;
if $e_y = 0$ **then return** ± 0;

$e_{spl} = \max(e_{x_{spl}}, e_{y_{spl}})$;
if $e_{spl} = 254$ **then**
  if $e_x = 255$ **then**
    if $f_x \neq 0$ **then return** x;
    if $e_y = 255$ **and** $f_y \neq 0$ **then return** y;
    if $e_y = 0$ **then return** NaN;
    return $\pm\infty$;
  if $e_x = 0$ **then**
    if $e_y = 255$ **then return** NaN;
    return ± 0;
  return $\pm y$;

Figure 6.1: Basic pseudo-code on the left, and an optimized way on the right for handling special values.

6.3 Product of Mantissas

In this section we first present how product of the mantissas is obtained. Next, we discuss how different methods influence the implementation of mantissa multiplication.

As said in case of floating-point addition, when an operand is a subnormal number ($e = 0$ and $f \neq 0$), the exponent of such operand is set to $e = 1$ and $m = 0.f$. Since the mantissa of a subnormal number has leading zeros, the product will also have leading zeros. If the tentative exponent $e_z \leq 0$, then the result is a subnormal number and gradual underflow occurs. If $e_z > 0$, the result may be a normalized number. In this case, the product is shifted left (while decrementing the exponent) until either it becomes normalized or $e_z \leq 0$. Generally in implementation, the mantissa of a subnormal operand is normalized and the exponent is decremented correspondingly. It allows to always have a product with a leading one. Indeed, if the decremented exponent causes $e_z \leq 0$, then the returned result is either zero or a subnormal number.

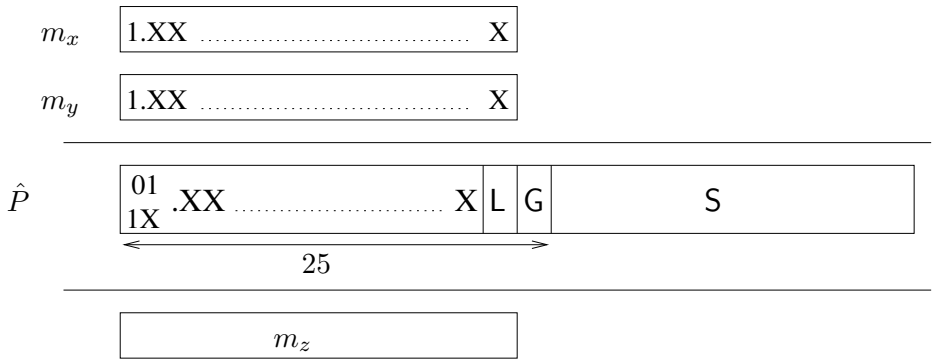


Figure 6.2: Multiplication of two 24-bit mantissas.

Unlike floating-point addition where three extra bits are required to obtain correctly rounded results, in the case of floating-point multiplication two extra bits suffices. The two extra bits, guard G and sticky S are determined from the 48-bit product obtained by multiplying two 24-bit mantissas as shown in Figure 6.2. The product is computed as

$$\hat{P} = p_{47}p_{46}.p_{45}\ldots p_{25}p_{24}p_{23}p_{22}p_{21}\ldots p_1p_0.$$

Since only 24 bits are required in the final result, guard and sticky bits are computed from the remaining 24 bits. Since $1 \leq m_x, m_y < 2$, $1 \leq \hat{P} < 4$ and thus it resembles any of the following two configurations.

$$\hat{P} = \begin{cases} 01.p_{45} \ldots p_{23}p_{22}p_{21} \ldots p_1p_0 \\ 1X.p_{45} \ldots p_{23}p_{22}p_{21} \ldots p_1p_0. \end{cases}$$

If $p_{47} = 0$, $\hat{P}$ is already normalized and $\mathsf{L} = p_{23}$, $\mathsf{G} = p_{22}$ and $\mathsf{S} = \mathrm{OR}(p_{21}p_{20} \ldots p_1p_0)$. If $p_{47} = 1$, $\hat{P}$ is normalized by shifting right by 1 bit and $\mathsf{L} = p_{24}$, $\mathsf{G} = p_{23}$ and $\mathsf{S} = \mathrm{OR}(p_{22}p_{21} \ldots p_1p_0)$.

Once the product has been normalized, if necessary, rounding is performed as in floating-point addition, but now with only two additional bits.

6.4 Rounding

In this section we provide the rounding function for each rounding mode as in floating-point addition. Rounding requires an addition of a bit $rnd \in \{0, 1\}$ to L and if it causes an overflow, it is necessary to normalize again by incrementing the exponent. The Table 5.3 shows the computation of rnd. After the rounding is performed the mantissa of the result must be obtained.

The product $\hat{P}$ can be partitioned into two parts as

$$\hat{P} = \underbrace{p_{47}p_{46}.p_{45} \ldots p_{25}p_{24}p_{23}}_{\hat{P}_{upper}} \underbrace{p_{22} \ldots p_1p_0}_{\hat{P}_{lower}}.$$

The result mantissa is computed as

$$m_z = \begin{cases} \hat{P}_{upper} + rnd & \text{if } p_{47} = 0, \\ (\hat{P}_{upper} >> 1) + rnd & \text{if } p_{47} = 1. \end{cases} \tag{6.1}$$

6.5 Optimizations

6.5.1 Special Values and Trivial Computations

The approach of handling the special values is exactly the same as for floating-point addition in Section 5.2. Also, the trivial computation step has been included which gets rid of all zero operands in the beginning.

6.5.2 Product of Subnormal Mantissas

In order to obtain a correct product one must know *in advance* whether any of the operand is a subnormal number. Let us say, x is a subnormal number. Now, the product $\hat{P}$ will be computed as $\hat{P} = 0.f_x \times m_y$. One comparison must be done *in the beginning* to decide whether to reinsert the implicit bit or not. This comparison is expensive, specially when x being a subnormal is very rare. Also, $\hat{P}$ might have leading zeros and detection of leading one might be expensive. We try to avoid this comparison in the following way.

1. Treat all the operands as normalized numbers.
2. Reinsert the implicit bit while forming the mantissas.
3. Compute the product as $\hat{P} = 1.f_x \times m_y$.

4. In special values verification, since $e_x = 0$, this particular case will be trapped. While checking e_x for zero, modify the product as $\hat{P} = \hat{P} - m_y$.

This is due to the fact that if we consider $m_x = 1 + f_x$ and $m_y = 1 + m_y$ then the product can be written as $\hat{P} = 1 + f_y + f_x + f_x f_y$. Since $m_x = 0 + f_x$, $\hat{P} = f_x + f_x f_y$ which is $\hat{P} - m_y$. This optimization allows to completely avoid the rare cases to influence the main execution path. Implementation of this technique only requires a shift followed by an addition.

6.5.3 Post-Normalization

If $\hat{P} = 1.11\ldots1$ after normalization, it will become $\hat{P} = 10.00\ldots0$ after rounding. Normalization is required again. We try to detect this normalization and perform it before rounding. There are two reasons for doing this.

1. Detection can be done by checking the bit pattern of $\hat{P}$ after normalization.

2. Since the representation uses hidden bit, it does not affect the result mantissa. Only the result exponent is modified and it can be done in parallel.

6.6 Performances

Performances in Section 3.5 give general timings for multiplication for different types of support. In this section we will discuss how these timings are obtained and for which input values. As we studied for floating-point addition, the impact of different types of values in different cases on general timings, we do the same here.

The general timings from Section 3.5 for multiplication are given in the Table 6.2 below. Recalling the supports A, B, C, and D as were defined in Section 5.6 for floating-point addition.

	A	B	C	D
ST231	36	45	52	60
ST220	38	47	54	62

Table 6.2: General timings for floating-point multiplication on ST220 and ST231.

As discussed in Section 6.3 that with the $32 \times 32 \rightarrow 32$ multiplier available on ST231 the product of the mantissas can be obtained in two parallel multiplications instead of four in the case of ST220. Thus performing floating-point multiplication is always 2 cycles faster on ST231 than on ST220 as shown by the timings.

6.6.1 Influence of the Trivial Computation Step

Table 6.2 shows, what happens to floating-point multiplication of input operands which are normalized numbers, when trivial computation step is included.

x, y	A	B	C	D
$7, 11$ (without)	31	47	47	62
$7, 11$ (with)	38	47	54	62

6.6.2 Different Input Values

Special values

$e_x, e_y \neq 255$	A	B	C	D
$0, y$	17	32	17	32
$x, 0$	17	37	17	37
$0, 0$	17	32	17	32

x, y	A	B	C	D
$\infty/\text{NaN}, y$	30	24	30	24
$x, \infty/\text{NaN}$	30	38	30	38
$\infty/\text{NaN}, \infty/\text{NaN}$	30	24	30	24
$\infty, 0$	34	24	34	24
$0, \infty$	34	32	34	32

Subnormal values

x, y	A	B	C	D
x_{sub}, y	17	73	17	88
x, y_{sub}	17	67	17	82
x_{sub}, y_{sub}	17	77	17	77

Specific case: flush to zero

x, y	A	B	C	D
$2^{-126}, 1 - 2^{-24}$	38	50	57	69

6.6.3 Input Values Causing Overflow/Underflow

x, y	A	B	C	D
(overflow)	38	47	60	62
(underflow)	38	50	54	69

6.6.4 Handling Special Values

Basic way: pseudo-code on the left in Figure 6.1.
Optimized way: pseudo-code on the right in Figure 6.1

x, y	A	B	C	D
$7, 11$	37	61	61	76

x, y	A	B	C	D
$7, 11$	40	52	57	72

6.6.5 Post-Normalization Step

As explained in Section 6.5 this step is performed completely independent of the rounding and can be done in parallel. The timings below shows the impact of advancing this step.

x, y	A	B	C	D
$2 - 2^{-23}, 1 + 2^{-23}$ (before rounding)	38	47	57	62
$2 - 2^{-23}, 1 + 2^{-23}$ (after rounding)	41	50	54	60

Chapter 7

Square

This operation is used in many graphics, telecommunication and/or control applications. The motivation behind supporting this operation in FLIP is that although square can be computed through multiplication, further optimization is possible. Since there is a single input operand, handling of special values is simple and straightforward. Also, it is not possible that rounding causes an overflow so post-normalization is not required. Because of all these reasons it is worth implementing this operation. Interestingly, one can see that due to further optimizations it was made possible to obtain a speed-up factor of 1.88 in comparison to the floating-point multiplication of the original library (with no support for square so implemented through multiplication) and 1.46 in comparison to the one in FLIP. In this chapter we have only detailed the differences which account for the above said improved latency.

7.1 Floating-Point Square

From the point of view of the specifications, the IEEE standard does not explicitly supports this operation. The implementation of this operation is almost completely based on multiplication. Also, there is not much difference in performances. So, we will only discuss about the basic differences in both the operations and will present the steps which can further be optimized.

The input for floating-point square is a single precision floating-point operand x represented by (s_x, e_x, f_x). The implementation of this operation supports no subnormals and only round-to-nearest-even mode. The output is a single precision floating-point number $z = \circ(x^2)$. The mantissa m_x is formed after the reinsertion of the implicit bit and a short description of floating-point square can be given as follows.

1. If x is $\pm\infty$, ± 0 or NaN, return the result as defined in Table 7.1.
2. Multiplication of the mantissas is performed and the product is obtained as $\hat{P} \leftarrow m_x \times m_x$.
3. The exponent of the result is computed as

$$e_z \leftarrow 2e_x - 127.$$

   On the target architecture, it requires a single *shift-and-add* instruction.

4. The sign of the result is always positive so $s_z = 0$.

5. Normalize $\hat{P}$, if necessary, and update the exponent.

6. Round the normalized product and obtain the result mantissa as $m_z \leftarrow \circ(\hat{P})$. Post-normalization can not occur.

7. Determine exceptions by verifying the exponent of the result.

Now, we discuss the implementation of only those steps which differ from floating-point multiplication.

7.2 Special Values

When the input operand is a special value the result is chosen from Table 7.1. The pseudo-code on the left in Figure 7.1 implements this table and the one on the right has been implemented in FLIP.

x	NaN	$+\infty$	$-\infty$	$+0$	-0
z	NaN	∞	∞	0	0

Table 7.1: All the possible predefined values of the square z when x is a special value.

The pseudo-code on the right in Figure 6.1 is an optimized way of handling the special values.

if $e_x = 255$ **then**
  if $f_x \neq 0$ **then return** x;
  if $f_x = 0$ **then return** $+\infty$;
if $e_x = 0$ **then return** $+0$;

if $e_x = 0$ **then return** $+0$;
if $e_x = 255$ **then return** $|x|$

Figure 7.1: Basic pseudo-code on the left and an optimized way on the right for handling special values.

7.3 No Post-Normalization Step

The square of the mantissa is produced in the same way as the product of the mantissas. As in floating-point multiplication, the square has two different configurations and the bit p_{47} decided whether to normalize or not. In order to perform rounding according to the round-to-nearest-even mode two extra bits guard G and sticky S are required.

We know that normalization is required after rounding only when $\hat{P} \in [2 - 2\text{ulp}(1), 2)$ where $2 - 2\text{ulp}(1) = \underbrace{1.11\ldots1111}_{\text{24bits}}|10$. We take $x = \sqrt{2}$ and try to obtain $\hat{P}$ for three different mantissas m_x^-, m_x, m_x^+.

$$\hat{P} = \begin{cases} 1.11\ldots1100|11 & m_x^- \\ 1.11\ldots1111|01 & m_x \\ 10.00\ldots0001|01 & m_x^+ \end{cases}$$

It can be clearly seen that for the first two cases $\hat{P} \in [2-2\text{ulp}(1), 2-2\text{ulp}(1))$ and for the third case $\hat{P} \in [2, 4)$. Ignoring the third case since normalization is done once, either after or before

the rounding. Rounding the first two cases does not cause overflow and thus post-normalization is not required. Of course, if the rounding mode is positive or negative infinity post-normalization is required since the sticky bit is not zero.

Chapter 8

Fused Multiply-and-Add

FMA is an acronym used for fused multiply-and-add. This operation is important in many scientific and engineering applications. Floating-point units of several current processors execute this operation as a single instruction as $\pm x \times y \pm w$. This operation has been implemented in the floating-point units of the IBM RS/6000 [51], IBM POWER2 [96] and Intel Itanium [85] processors. The advantage of fused implementation is that rounding is done only once instead of twice (after multiplication and then after addition). Using the FMA instruction, correctly rounded extended precision floating-point operations can be efficiently implemented [45]. Also, this instruction is useful in software implementations of floating-point division, square root and remainder [22]. The FMA is also valuable in speeding-up and improving the accuracy of the computations involving accumulations of dot products. The revision of the IEEE standard, that is, IEEE 754r [6] will have a support for this operation. The motivation behind providing the support for this operation in FLIP is the statistics obtained after the analysis of two applications. More than 87% of additions and multiplications performed in the application SBC decoder (see Section 4.5) are associated together to produce a multiply-and-add operation. For another application DTS-HD decoder it is 75%. Our implementation is based on the implementations of floating-point multiplication and addition as described in Section 8.3. The most difficult part is the handling of special values which is described in Section 8.2. The support for FMA has allowed to compute $xy+w$ 2.21 times and $xy-z$ 2.18 times faster than when they were computed in the original library which does not support these operations. Hence, the original library is slower as well as less accurate since it supports them through two consecutive floating-point operations.

8.1 Floating-Point FMA

The operation fused multiply-and-add ($x \times y + w$) can also be used to perform floating-point multiplication only, by setting $w = 0$ or floating-pointaddition/subtraction only by setting, for example, $y = 1$. The three variants of this operation are

1. $x \times y - w$, called fused multiply-and-subtract (FMS),
2. $-x \times y + w$, called fused negative multiply-and-add (FNMA), and
3. $-x \times y - w$, called fused negative multiply-and-subtract (FNMS).

FMS can be computed in the same as floating-point subtraction by negating the sign bit of w and then performing FMA. The other variants can easily be performed through FMA.

The input for the floating-point FMA is three single precision floating-point operands x, y, and w represented by (s_x, e_x, f_x), (s_y, e_y, f_y) and (s_w, e_w, f_w) respectively. The implementation of this operation supports no subnormals and only round-to-nearest-even mode. The output is a single precision floating-point number $z = \circ(x \times y + w)$. Mantissas are constructed after the reinsertion of the implicit bit. A short description can be given as follows.

1. In case of a special value refer to Section 8.2.
2. The mantissas m_x and m_y are multiplied to produce the 48-bit product P. The exponents e_x and e_y are added and $d = e_x + e_y - 127 - e_w$ is computed. The tentative result exponent is $e_z = \max(e_x + e_y - 127, e_w)$.
3. The product P and the mantissa m_w are added as in floating-point addition.
4. The sum is normalized, if necessary, and the exponent e_z is updated.
5. The sum is rounded to nearest.
6. Determine exceptions by verifying the result exponent.

8.2 Special Values

Handling of special values is not different from what was done for floating-point addition or multiplication but is difficult since now, there are three input operands and the intermediate product must also be considered. Tables 8.1 and 8.2 below provides all the possible results if any of the input operands is a special value or if the intermediate result turns out to be a special value.

Since the original library does not support the FMA operation, we only provide the pseudo-code, in Figure 8.1, implemented in FLIP. See Section 5.5.2.1 to know why $e_{x_{spl}}$ and $e_{y_{spl}}$ are required and how they are computed ($e_{w_{spl}}$ is computed in the same way). The sign and the exponent of the product are indicated by s_{xy} and e_{xy} respectively.

Table 8.1 is exactly the same as for floating-point multiplication. Since the final result depends on the product $P = xy$, it is found first and Table 8.2 is consulted. But in some cases it is possible to return the final result without knowing w. For example, if $x = \pm 0$ and $y = \pm\infty$ the final result is NaN.

$\times$		y					
		NaN	$+\infty$	$-\infty$	$+0$	-0	y
x	NaN	NaN	NaN	NaN	NaN	NaN	NaN
	$+\infty$	NaN	P	P	NaN	NaN	P
	$-\infty$	NaN	P	P	NaN	NaN	P
	$+0$	NaN	NaN	NaN	P	P	P
	-0	NaN	NaN	NaN	P	P	P
	x	NaN	P	P	P	P	P

Table 8.1: All the possible predefined values of the product $x \times y$ when x or y is a special value.

Note that the implementation of this operation employs the trivial computation step for the early exit in case of zero operands. The pseudo-code for this step is given in Figure 8.2. Indeed,

$e_{spl} = \max(\max(e_{x_{spl}}, e_{y_{spl}}), e_{w_{spl}})$;
if $e_{spl} = 254$ **then**
  if $e_w = 255$ **and** $f_w \neq 0$ **then return** NaN;
  if $e_x = 255$ **then**
    if $f_x \neq 0$ **then return** NaN;
    if $e_y = 0$ **then return** NaN;
    if $e_y = 255$ **and** $f_y \neq 0$ **then return** NaN;
    if $e_w = 255$ **then**
      if $f_w \neq 0$ **then return** NaN;
      if $s_{xy} \neq s_z$ **then return** NaN;
    return $\pm\infty$;
  if $e_x = 0$ **and** $e_y = 255$ **then return** NaN;
  if $e_y = 255$ **then**
    if $f_y \neq 0$ **then return** NaN;
    if $e_w = 255$ **then**
      if $f_w \neq 0$ **then return** NaN;
      if $s_{xy} \neq s_z$ **then return** NaN;
    return $\pm\infty$;
    if $e_w = 255$ **then**
  if $f_w \neq 0$ **then return** NaN;
  if $e_{xy} = 255$ **and if** $s_{xy} \neq s_z$ **then return** NaN;
  return $\pm\infty$;

Figure 8.1: The implemented pseudo-code for handling special values.

$+$		P				
		$+\infty$	$-\infty$	$+0$	-0	P
w	NaN	NaN	NaN	NaN	NaN	NaN
	$+\infty$	$+\infty$	NaN	$+\infty$	$+\infty$	$+\infty$
	$-\infty$	NaN	$-\infty$	$-\infty$	$-\infty$	$-\infty$
	$+0$	$+\infty$	$-\infty$	$+0$	$+0$	P
	-0	$+\infty$	$-\infty$	$+0$	-0	P
	w	$+\infty$	$-\infty$	w	w	$\circ(P + w)$

Table 8.2: All the possible predefined values of the sum $\circ(P + w)$ when P or z is a special value.

this step induces a latency overhead of 7 cycles when this operation is measured for a single set of input values. But a significant percentage of trivial operands in both the applications (see Section 4.5) suggests the inclusion of this step.

8.3 Implementation and Optimizations

As said earlier, this operation is based on floating-point multiplication and addition. The multiplication $m_x \times m_y$ is done in exactly the same way as for floating-point multiplication. The difference here is that the exact product P (48 bits) is obtained and thus no sticky bit is required. Since the product can not be stored on a single 32-bit register, two 32-bit registers are used. One can

if $e_x = 0$ **or if** $e_y = 0$ **then**
  if $e_x \neq 255$ **and if** $e_y \neq 255$ **then**
    if $e_z = 0$ **then return** ± 0
    return z

Figure 8.2: The pseudo-code for trivial computation step.

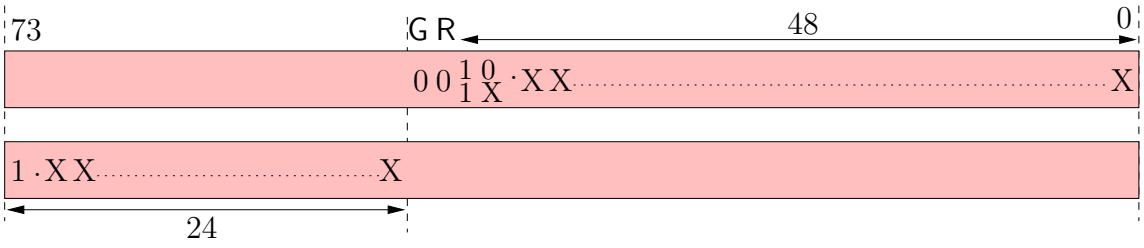


Figure 8.3: Position of the product and the mantissa as in the standard implementation.

imagine that P has been stored as a 64-bit value but the complicacies involved in representing values on two separate 32-bit registers and performing computations with these values has been completely hidden by the compiler. The full 48-bit product is used in addition. The third mantissa m_w is also stored as a 64-bit value. All this is depicted in Figure 8.4. Note the *optimized way* of positioning these two values. The standard way of positioning is depicted in Figure 8.3. The standard way allows to avoid the alignment shift for product, and the bidirectional shift of the mantissa. The mantissa has been positioned in advance 27 bits to the left of the product and is shifted only to right by $d = e_x + e_y - 127 - e_w + 27$. On the other hand, if $d \leq 0$, no alignment is required. The two zero bits shown in the figure are the guard and round bits when no alignment is done, that is, the result is m_w.

This way can not serve in our case for the following reasons.

1. For alignment, the product can be shifted with as much ease as the mantissa and in any direction.

2. Adopting the standard way would mean representing these two values as a 74-bit value. Since representation on 64 bits is already expensive, we have not tried positioning these values according to the standard way.

3. Our way allows to be close to the floating-point addition and thus the same implementation can be followed.

After swapping, if $d < 0$, and aligning by $|d|$, these two values are added by performing a 64-bit addition. The sum of this addition might resemble $1XX.XX\ldots X$, $1X.XX\ldots X$ or $0.00\ldots01X\ldots X$, in which case it must be normalized. Note that, even if the intermediate product is of the form $1X.XX\ldots X$, it is not normalized and is added as it is. That is why the form $1XX.XX\ldots X$. Because the carry might be generated, guard bits are maintained on the left. Once normalized, the most significant 26 bits are extracted with the computation of sticky bit from the discarded part. Rounding is performed for round-to-nearest mode as described in Section 5.4.

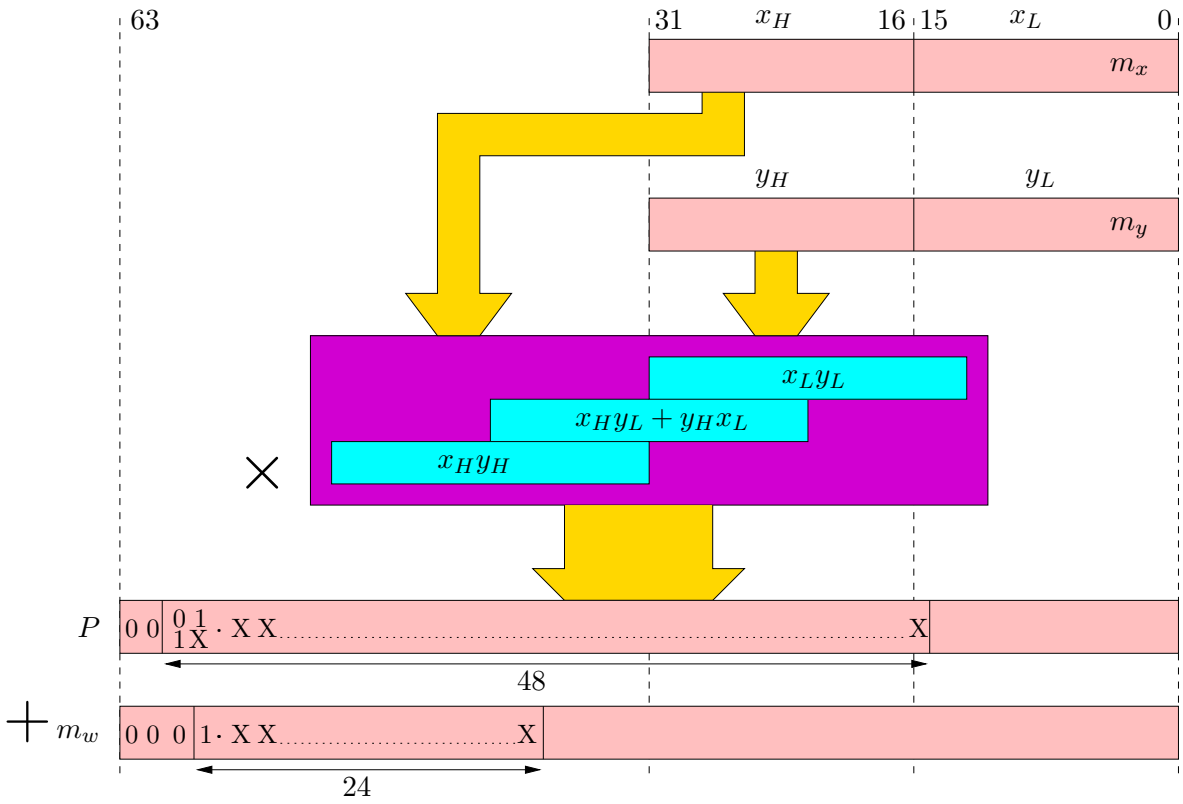


Figure 8.4: Illustration of the implementation of FMA (P can be obtained on both the target processors).

Part III

Division, Square Root and Related Operations

Chapter 9

Division

Among the basic arithmetic operations division is still the most complex and time consuming. A study from Oberman and Flynn [69] shows that even if the number of issued division instructions is low (around 3% for SPEC benchmarks), the total duration of the division computation cannot be neglected (up to 40% of the time in arithmetic units). So a fast implementation of division is indispensable. In hardware, a dedicated FP unit generally employs SRT or Newton-Raphson's algorithm using FMA to perform FP division. When implemented in software, which is our case here, it is generally based on nonrestoring, SRT radix-2, or Newton-Raphson's algorithm. This chapter recalls three different classes of division algorithms, namely digit recurrence, iterative and very-high radix, and focuses on their software implementations optimized for target processors. We are not aware of any implementation of division in software based on a very-high radix algorithm. This chapter also tries to discuss the complicacies involved in the implementation of different algorithms and to respond why a particular algorithm is ill/well suited to a particular architecture. Finally, a comparative analysis shows that the very-high radix algorithms can be adapted very efficiently to the ST220 processor due to its rectangular multiplier. Iterative algorithms indeed provide the best latency but are only well suited for the ST231 processor since a full-width multiplier is available.

9.1 IEEE Floating-Point Division

The inputs for division are

1. Two operands x and y as the *floating-point dividend* and the *divisor* represented by (s_x, e_x, f_x) and (s_y, e_y, f_y) respectively.

2. One of the four rounding modes to specify how to round the exact result.

The output is a floating-point number $z = \diamond(x/y)$, where $\diamond$ denotes the active rounding mode.

From now on, throughout this chapter, the *normalized mantissas* m_x and m_y (Section 1.8) shall be called the *dividend* and the *divisor* respectively. The basic description of floating-point division can be given as follows:

1. If any of the input operands is a special value, the result is selected from Table 9.1.
2. Mantissas are divided and the quotient $Q = m_x/m_y$ is computed.
3. The intermediate result exponent is computed as
$$e_z = e_x - e_y + B.$$
4. The sign of the result is computed as
$$s_z = s_x \text{ XOR } s_y.$$
5. If $m_x < m_y$, normalization is required and consequently the result exponent is updated as $e_z = e_z - 1$.
6. The exact quotient Q is rounded to fit in the target precision. In case of division, rounding never causes an overflow of the mantissa; thus post-normalization is not required.
7. Exceptions are determined by verifying the exponent of the result.

9.2 Algorithms and Implementation

This section explains how to perform and implement each step of the basic description. Special values are handled in the same way as for other basic operations (see Sections 5.2 and 6.2). After this, we recall the basic definition of division in Section 9.2.2 followed by the three different classes of algorithms [32, 33] in Section 9.2.3. Finally, we discuss the problem of correct rounding for each class of division algorithm. After all this, the rest of the chapter concerns the implementation of different division algorithms tuned to IEEE standard and optimized for target processors.

9.2.1 Special Values

As for other basic operations, the philosophy of handling special values is the same for division too. Result is selected from Table 9.1 which displays all the possible results when x or y is a special value.

$/$		y					
		NaN	$+\infty$	$-\infty$	$+0$	-0	y
x	NaN	NaN	NaN	NaN	NaN	NaN	NaN
	$+\infty$	NaN	NaN	NaN	$+\infty$	$-\infty$	$\pm\infty$
	$-\infty$	NaN	NaN	NaN	$-\infty$	$+\infty$	$\pm\infty$
	$+0$	NaN	$+0$	-0	NaN	NaN	± 0
	-0	NaN	-0	$+0$	NaN	NaN	± 0
	x	NaN	± 0	± 0	$\pm\infty$	$\pm\infty$	$\diamond(x/y)$

Table 9.1: All the possible predefined values of the division $\diamond(x/y)$ when x or y is a special value.

The pseudo-code of handling special values is provided in Figure 9.1, the basic approach on the left and the optimized one on the right. Refer to Section 5.5.2.1 for complete details on precomputation of values such as e_{spl} and $e_{x_{spl}}$ and the importance of the one cycle `MAX` instruction here.

```
if e_x = 255 then
  if f_x ≠ 0 then return x;
  if e_y = 255 then return NaN;
  return ±∞;
if e_y = 255 then
  if f_y ≠ 0 then return y;
  return ±0;
if e_x = 0 then
  if e_y = 0 then return NaN;
  return ±0;
if e_y = 0 then return ±∞;
```

```
e_spl = max(e_{x_spl}, e_{y_spl});
if e_spl = 254 then
  if e_x = 255 then
    if f_x ≠ 0 then return x;
    if e_y = 255 then return NaN;
    return ±∞;
  if e_y = 255 then
    if f_y ≠ 0 then return NaN;
    return ±0;
  if e_x = 0 then
    if e_y = 0 then return NaN;
    return ±0;
  if e_y = 0 then return ±∞;
```

Figure 9.1: Basic pseudo-code on the left and an optimized way on the right for handling special values.

9.2.2 Definitions

The problem of the division of the mantissas is: given $1 \leq m_x, m_y < 2$ as above, compute the quotient $Q \in (1/2, 2)$ and, optionally, the remainder Rem such that

$$m_x = Q \times m_y + Rem \tag{9.1a}$$

and

$$|Rem| < |m_y| \times \text{ulp} \quad \text{and} \quad sign(Rem) = sign(m_x). \tag{9.1b}$$

The n-bit quotient is defined to comprise k radix-r digits with

$$r = 2^b \tag{9.2}$$

$$k = \frac{n}{b} \tag{9.3}$$

where a division algorithm that retires b bits of quotient in each iteration is said to be a radix-r algorithm. Such an algorithm requires k iterations to compute a quotient of n bits. The precision of the quotient is defined by the unit in the last position (ulp), where for an integer quotient $\text{ulp}(Q) = 1$, and for a fractional quotient $\text{ulp}(Q) = r^{-n}$, assuming a radix-r representation and n-bit quotient. Note that the radix of the algorithm and the radix of the floating-point representation are independent quantities and need not be the same.

9.2.3 Division of Mantissas

There exist many different methods for dividing the mantissas [33, p. 247]. They have been classified into three main classes as follows:

1. *Digit recurrence* algorithms (see Section 9.3): one quotient digit per iteration using a residual recurrence which involves addition, multiplication by a single digit (generally performed through a shift), and a quotient-digit selection function.

    - Produces both the quotient, which can easily be rounded, and the remainder.

- The sticky bit, useful in rounding, is provided by the condition "remainder not equal to zero".

2. *Very-high radix* algorithms (see Section 9.4): similar to digit recurrence algorithms but as the radix increases, the quotient digit selection function and generation of the divisor multiples become more complex. So a simpler selection function is required. The initial step of the algorithm consists of computing an initial approximation of the reciprocal of the divisor, accurate up to a fixed number of bits, and scaling both the dividend and the divisor by multiplying them with this approximation. This scaling of the dividend and the divisor is called the prescaling step. The final step performs few iterations (the number of iterations depends on the accuracy of the approximation) of *quotient-selection by rounding* and produces the partial remainder(s) through multiplication and addition/subtraction.

3. *Functional iteration* algorithms (see Section 9.5): these are quadratically convergent algorithms that double the number of bits of the quotient after each iteration through floating-point multiplications. These methods do not produce directly the truncated quotient required for rounding [33, p. 457]. So, correct rounding is difficult as it requires either the computation of extra quotient digits (increasing the number of iterations) or the computation of all the intermediate steps in larger precision (increasing the latency of each iteration).

9.2.4 Correct Rounding of the Quotient

Since the implementation is tuned to the IEEE 754 standard, we finally need a 24-bit mantissa of the result as $m_z = \diamond(Q)$. A 25-bit quotient Q is obtained and is normalized, if required, such that $1 \leq Q < 2$, bit 25 being the round bit. The sticky bit is needed for rounding to plus and to minus infinity and is provided by the condition "remainder not equal to zero". However, it is not needed for round to nearest, since the tie condition can not occur as explained in [33, p. 452]. Computation of the round bit depends on the particular class of algorithm used to obtain Q. For the digit recurrence algorithms, Q is directly constructed by a fixed number of bits produced at each iteration of the recurrence. For very-high radix algorithms, the same iteration as in the digit recurrence algorithms is performed through multiplication instead of a simple shift. In these iterations truncated multiplications are used and so the computed quotient converges toward the exactly rounded result, but it may be either above or below it [69]. Consequently, the sign of the remainder indicates the position of the computed quotient relative to the exact quotient. Functional iteration algorithms result in approximations that do not produce directly the quotient required for rounding. To determine this quotient it is necessary to obtain

$$Rem_{temp} = m_x - Q \times m_y.$$

If $Rem_{temp} < 0$, Q is reevaluated as $Q = Q - \text{ulp}(Q)$.

9.3 Digit Recurrence Algorithms

The most common division algorithms implemented in processors today belong to the class of "digit recurrence algorithms". They all share the same way of producing a quotient: perform a certain number of iterations and retire a fixed number of bits of the quotient in each iteration. The well known, shift-and-subtract algorithm that everybody learns in elementary school, is actually a digit recurrence algorithm. What we learnt in school is a radix-10 division algorithm which means that each quotient digit can be chosen from a set of ten values. When these algorithms

are implemented in order to use with computers the choice of representing the radix a power of 2 would be better.

In digit recurrence algorithms every iteration produces one quotient digit, most significant quotient digit first. Implementation of such algorithms is greatly influenced by the choice of radix, the choice of allowed quotient digits, and the representation of the partial remainder (or residual). The radix determines the number of quotient bits retired in an iteration, which in turn, determines the division latency. Increasing the radix helps to reduce the number of iterations (reduce the latency), but increases the time for each iteration so overall latency is not reduced as expected [69]. A careful choice of the allowed quotient digits can reduce the time for each iteration, but with a corresponding increase in the complexity of the selection function. Similarly, different representations of the residual can reduce the iteration time but the selection function becomes more complicated. Generally, in hardware implementation, a redundant representation of the residual is used which allows to select a quotient digit based on the estimation (few most significant bits) of the residual. This representation reduces the addition/subtraction time involved in each iteration. But in software implementation, which is our case here, addition/subtraction time is fixed and so we use a full residual instead of an estimation. Hence, different representations of the residual do not serve in our case.

The most basic digit recurrence algorithms are *restoring* (see Section 9.3.2) and *nonrestoring* division (see Section 9.3.3). The most commonly implemented digit recurrence algorithms belong to the *SRT family of division algorithms*. This family of division algorithms was named after D. Sweeney of IBM [19], J. E. Robertson of the University of Illinois [77], and T. D. Tocher of the Imperial College of London [93], each of whom developed it independently around the same time. The idea of using the redundant representation for the residual was introduced by D. E. Atkins [16], and K. G. Tan [92] gave a theory of implementing high radix SRT division based on look-up tables. Since the inception of SRT division, it has been widely explored and well researched. Ercegovac and Lang have dedicated a complete book on division and square root based on digit recurrence algorithms and particularly, SRT algorithms [32]. The division algorithm implemented on the Pentium chip that caused the appearance of the "floating-point division bug" in 1994 was radix-4 SRT algorithm. The quotient-digit selection function of this SRT algorithm was based on some look-up tables. Due to five incorrect entries in these tables, floating-point division on the Pentium chip seldom produced less-precise quotients. A complete review about the existence and the remedy of this bug has been given in [21, 28].

In case of digit recurrence algorithms, rounding is performed by computing the round bit of the quotient and examining the final remainder. An extra iteration might be required to compute the round bit. One ulp (of the quotient) is conditionally added according to the specified rounding mode and depending on the round bit and the remainder. Correct rounding requires the determination of the sign of the final remainder and whether the final remainder is exactly zero. To obtain n significant bits of the quotient, n bits are computed if the leading bit is 1 (in case when $m_x \geq m_y$), and $n+1$ bits if it is 0 (in case when $m_x < m_y$).

We begin by a discussion of the implementation issues concerning the digit recurrence algorithms in Section 9.3.1. We present the restoring and nonrestoring division algorithms in Sections 9.3.2 and 9.3.3 respectively. The rest of this section is dedicated to the philosophy of SRT division algorithms followed by the implementation issues involved in the radix-2 and the radix-4 SRT algorithms in Sections 9.3.4 and 9.3.5.

9.3.1 Implementation of Digit Recurrence Algorithms

In general, to simplify the presentation of the algorithms, we modify the general definitions and assume $1/2 \leq m_x < m_y$ and $m_y \in [1, 2)$. Hence $Q \in (1/4, 1)$. The quotient $Q = \sum_{j=1}^{n} q_j r^{-i}$ is supposed to be an n-bit number with the q_j belonging to the digit set as described in Section 9.3.1.1. We present three main aspects of digit recurrence algorithms and the way these aspects influence the decision to consider a particular algorithm.

9.3.1.1 Choice of Quotient Digit Set

The simplest quotient digit set for radix r is to have exactly r allowed values of the quotient. The digit set for the restoring algorithm is $\{0, 1\}$ and for the nonrestoring algorithm it is $\{-1, 1\}$ which is a nonredundant digit set. In both os these algorithms residuals are compared to all the divisor multiples which is not an efficient way of selecting the quotient digits. So, to speed up the computation of q_j redundancy is introduced into the digit set. Now, q_j can be chosen among more than r values [32, p. 10]. Such a digit set can be composed of symmetric signed-digit consecutive integers, where the maximum digit is a. The digit set is made redundant by having more than r digits in the set. In particular,

$$q_j \in \mathfrak{D}_a = \{-a, -a+1, \ldots, -1, 0, 1, \ldots, a-1, a\},$$

where a must satisfy

$$a \geq \lceil r/2 \rceil.$$

The redundancy of a digit set is determined by the value of the redundancy factor ρ, which is defined as

$$\rho = \frac{a}{r-1}, \quad \rho \geq \frac{1}{2}. \tag{9.4}$$

Typically, signed-digit representations have $a < r - 1$. When $a = \lceil r/2 \rceil$, the representation is called *minimally redundant*, while that with $a = r - 1$ is called *maximally redundant*, with $\rho = 1$. A representation with digit set $\mathfrak{D}_a$ and radix r is, *incorrect* if $a < (r-1)/2$ and *nonredundant* if $a = (r-1)/2$. A representation where $a > r - 1$ is called *over-redundant*. Such a redundant digit set allows simplification of the quotient-digit selection function as described in Section 9.3.1.2.

9.3.1.2 Recurrence

The following recurrence is used in each iteration:

$$w[0] = m_x, \tag{9.5a}$$

$$w[j] = rw[j-1] - m_y \times q_j, \quad j = 1, \ldots, k, \tag{9.5b}$$

where $w[j]$ is the residual after j iterations and k is the number of iterations required to produce an n-bit quotient. From (9.5), assuming each iteration produces one bit of the quotient ($k = n$), we get

$$w[n] = m_x r^n - (q_1 r^{n-1} + q_2 r^{n-2} + \cdots + q_n) m_y.$$

Hence,

$$\frac{m_x}{m_y} - \underbrace{(q_1 r^{-1} + q_2 r^{-2} + \cdots + q_n r^{-n})}_{0.q_1 q_2 \ldots q_n} = \frac{w[n]}{m_y} r^{-n}.$$

This means that if $0 \leq w[n] \leq m_x r^n - (q_1 r^{n-1} + q_2 r^{n-2} + \cdots + q_n) m_y$ the number $Q = q_1 r^{-1} + q_2 r^{-2} + \cdots + q_n r^{-n}$ is the desired quotient.

After k iterations, a correct division algorithm must produce a quotient Q with an error $\epsilon_Q = \epsilon[n] \geq 0$, with respect to the infinite precision quotient value m_x/m_y of less than one ulp(Q) (see (9.1)). That is,

$$0 \leq \epsilon_Q = \frac{w[n]}{m_y} r^{-n} = \frac{m_x}{m_y} - Q < r^{-n}. \tag{9.6}$$

If all the intermediate residuals $w[j]$ are computed correctly, i.e. $\epsilon[j] < r^{-j}$, then the above recurrence should converge to this error. So it is necessary to bound the error after every iteration. Let us call $Q[j]$ the value of the quotient and $\epsilon[j]$ the error after j iterations, then a possible bound can be given as

$$\epsilon[j] = \left| \frac{w[j]}{m_y} r^{-j} \right| = \left| \frac{m_x}{m_y} - Q[j] \right| < r^{-j}. \tag{9.7}$$

This ensures that the magnitude of the error committed after n iterations is bounded as in (9.6). For the IEEE single precision format, which is our case here, we have the error $|\epsilon[n]| < 2^{-23}$. However, this error can be negative; in such a case, an additional correction step is required which modifies both $w[n]$ and Q to make ϵ_Q positive. From (9.7), a bound on the residual can easily be deduced, which is

$$|w[j]| < m_y \tag{9.8}$$

and

$$|w[j]| < \rho \times m_y \tag{9.9}$$

when a redundant quotient digit set is used. In each iteration, one digit of the quotient is determined by the quotient-digit selection function

$$q_j = SEL(w[j-1], m_y) \tag{9.10}$$

and, in order to have a correct algorithm, the value of the quotient digit is selected such that the next residual is bounded by (9.8) or (9.9). All digit recurrence algorithms implement this selection function based on the intermediate residuals, produced from the recurrence (see (9.5)), and the divisor. We will see later that in order to reduce the latency, different algorithms employ different techniques to keep the selection function as simple as possible. Use of redundant digit set allows the comparisons to be done with *limited* precision constants only. In hardware, it is done using a table addressed by a few most significant bits of the divisor m_y and the residual $w[j+1]$. As said earlier in Section 9.3, the residual is generally represented using a redundant notation to speedup the subtraction of the iteration in (9.5). In *software*, which is our case here, *residual* is represented in *full precision nonredundant form*. Also, tables for implementing the selection function should not be used in order to avoid cache misses when the size of the table increases. When a redundant digit set is used there exists a tradeoff for selection function. We will see in case of radix-4 SRT algorithm that by using a large number of allowed quotient digits a, and thus a large value for ρ, the complexity of the quotient-digit selection function can be reduced. But, for every quotient digit, a multiple of the divisor needs to be generated. Multiples that are powers of two can be formed by a simple shift. If a multiple is required that is not a power of two (e.g., three), an addition may also be required which complicates the situation. So, maintaining an equilibrium between the complexity of the selection function and the generation of the divisor multiples is difficult.

The remainder is obtained from the last residual as follows:

$$Rem = \begin{cases} w[n]r^{-n} & \text{if } w[n] \geq 0, \\ (w[n] + m_y)r^{-n} & \text{if } w[n] < 0. \end{cases}$$

Moreover, if $w[n] < 0$ then $Q = Q - r^{-n}$.

9.3.1.3 Choice of Radix

Generally, increasing the radix decreases the number of iterations required for the same quotient precision, thus decreases the overall latency of the algorithm. The SRT algorithm in radix $r = 2$ produces one digit (one bit) of quotient in each iteration. In radix $r = 2^b$, each iteration produces one digit (b bits) of quotient, thus reduces the number of iterations to

$$k = \lceil n/b \rceil, \tag{9.11}$$

where $\lceil \alpha \rceil$ is the usual ceiling function of a real number α. Unfortunately, as the radix increases, every iteration becomes more complicated and the overall latency is not reduced as expected. Choosing the radix a power of 2 allows to compute the product $rw[j]$ of the radix and the residual by a simple shift. The simplest quotient digit set for radix r is $\{0, 1, 2, \ldots, r-1\}$, that is, to have exactly r allowed values. So again, it becomes impractical to generate all the required divisor multiples $(0, m_y, 2m_y, \ldots, (r-1)m_y)$ for higher radices.

9.3.2 Restoring Division

The restoring algorithm [32, p. 175] uses radix $r = 2$ with quotient digit set $\{0, 1\}$. At each iteration, the algorithm subtracts the divisor m_y from the previous residual $w[j-1]$ multiplied by 2 (line 4 in Figure 9.2). The quotient-digit selection function is derived by comparing the residuals at each step with the divisor multiples (here 0 or m_x). If the result is strictly less than 0, the previous value should be restored (line 9 in Figure 9.2). Usually, this restoration is not performed using an addition, but by selecting the value $2w[j-1]$ instead of $w[j]$, which requires the use of an additional register. An initialization step (not shown in the figure) ensures that the first bit of the quotient is 1 and thus $n = 24$ iterations are required to compute the remaining 23 bits plus the round bit.

```
1  w[0] ⟵ m_x
2  for j from 1 to n do
3      w[j] ⟵ 2 × w[j − 1]
4      w[j] ⟵ w[j] − m_y
5      if w[j] ≥ 0 then
6          q_j ⟵ 1
7      else
8          q_j ⟵ 0
9          w[j] ⟵ w[j] + m_y
```

Figure 9.2: Restoring division algorithm

The restoring algorithm requires n shifts, n subtractions and $n/2$ additions (on average) to obtain n bits of the quotient (see [32, p. 179]).

9.3.3 Nonrestoring Division

Nonrestoring division [32, p. 179] is an improved version of the restoring division in the sense that it completely avoids the restoration step by combining restoration additions with the next recurrence, thus reducing the overall latency. Moreover, it uses the quotient digit set $\{-1, +1\}$ to perform directly the recurrence with the selection function

$$q_{j+1} = \begin{cases} +1 & \text{if } w[j] \geq 0, \\ -1 & \text{if } w[j] < 0. \end{cases}$$

In this way, small errors in one iteration can be corrected in subsequent iterations. The new modified version presented in Figure 9.3 allows the same small amount of computations at each iteration. The conversion of the quotient from the digit set $\{-1, 1\}$ to the standard set $\{0, 1\}$ can be done *on the fly* by using the following algorithm (see [33, p. 256]):

$$Q[j] = \begin{cases} Q[j-1] + q_j 2^{-j} & \text{if } q_j \geq 0, \\ Q[j-1] - |q_j| 2^{-j} & \text{if } q_j < 0. \end{cases}$$

If $w[n] < 0$, the divisor is added to it and the quotient is modified in order to ensure a positive final remainder. Again, an initialization step ensures $q_1 = 1$ and so $n = 24$ iterations are required.

```
1  w[0] ⟵ m_x
2  for j from 1 to n do
3      w[j] ⟵ 2 × w[j − 1]
4      if w[j] ≥ 0 then
5          w[j] ⟵ w[j] − m_y
6          q_j ⟵ 1
7      else
8          w[j] ⟵ w[j] + m_y
9          q_j ⟵ −1
```

Figure 9.3: Nonrestoring division algorithm

The nonrestoring algorithm requires n shifts and n additions/subtractions to obtain an n digit quotient, and is therefore faster than the restoring one (see [32, p. 180]).

9.3.4 Radix-2 SRT Division

In the nonrestoring algorithm, an addition or a subtraction is always performed, whether $q_j = -1$ or $q_j = +1$. Radix-2 SRT division [32, p. 36] (Figure 9.4) uses a redundant quotient digit set $\{-1, 0, 1\}$ and selects $q_j = 0$ that replaces some additions/subtractions by simple shifts.

The same recurrence as in (9.5) will be used with the radix $r = 2$ but the difference lies in the implementation of the selection function. The fundamental change brought by SRT algorithms was to compare the residuals (represented in full precision nonredundant form) with the *limited* precision constants only. This can be done by restricting the range of residuals to $[-1/2, 1/2)$ rather than to $[-m_y, m_y)$. This restriction can be enforced by assuming $m_y \geq 1/2$ and, because of this, one may have to shift the initial residual, that is, $w[0] = m_x$ in order to respect the bound in (9.9). For quotient digit set $\{-1, 0, 1\}$, from (9.9) we compute $\rho = 1$ and so the quotient digit must be chosen such that

$$|w[j]| < m_y.$$

```
1  w[0] ⟵ m_x
2  for j from 1 to n do
3      w[j] ⟵ 2 × w[j − 1]
4      if w[j] ≥ 1/2 then
5          w[j] ⟵ w[j] − m_y
6          q_j ⟵ 1
7      else
8          if w[j] < −1/2 then
9              w[j] ⟵ w[j] + m_y
10             q_j ⟵ −1
11         else
12             q_j ⟵ 0
```

Figure 9.4: Radix-2 SRT division algorithm

The radix-2 SRT algorithm uses the selection function

$$q_{j+1} = \begin{cases} +1 & \text{if } 1/2 \le w[j] \\ 0 & \text{if } -1/2 \le w[j] < 1/2 \\ -1 & \text{if } w[j] < -1/2. \end{cases}$$

Figure 9.5 is the P-D diagram which corresponds to the above selection function. The P-D diagram is useful when designing the quotient-digit selection function. It has as axes, the shifted Partial remainder and the Divisor. The selection interval bounds are defined by the different lines starting at the origin. The shaded region is the overlap region where more than one quotient digit may be selected. The amount of overlap is given by $(2\rho - 1)m_y$. Since the amount of overlap depends on ρ, higher redundancy in the quotient-digit set produces larger overlap regions which in turn allow more flexibility in the choice of the actual quotient digit. Moreover, a good amount of overlap exists when $1/2 \le m_y < 1$; hence the initial condition $m_y \ge 1/2$. Note that since the decision boundaries (dark lines in the overlap region) in the P-D diagram are straight, the selection function does *not* depend on m_y. Also, when the residual lies in $[-1/2, 1/2)$ then 0 is selected and thus no operation is performed. Again, as in the nonrestoring algorithm, the conversion of the quotient from the redundant digit set $\{-1, 0, 1\}$ to the standard digit set $\{0, 1\}$ can be done *on the fly* by using a simple algorithm (see [33, p. 256]). If $w[n] < 0$, the divisor is added to it and the last bit of the quotient is modified accordingly in order to ensure a positive final remainder.

9.3.5 Radix-4 SRT Division

In this SRT division [32, p. 41] the radix is increased to $r = 2^2$. So the new recurrence can be written as

$$w[j] = 4w[j-1] - D \times q_j, \quad j = 1, \ldots, k.$$

This recurrence allows to compute the required quotient in $k = \lceil n/2 \rceil$ iterations. For the radix-4 SRT algorithm, there exist two possibilities for choosing the redundant quotient-digit set. The digit sets are $\{-2, \ldots, 2\}$ when $a = 2$ and $\{-3, \ldots, 3\}$ when $a = 3$. Both digit sets have their own advantages and drawbacks. In general, choosing a bigger set allows to reduce the complexity of the selection function. But, for every quotient digit, a multiple of the divisor needs to be generated. So there exists a tradeoff between the complexity of the selection function and the difficulty in generating the multiples. When $a = 2$, four multiples $(m_y, 2m_y, -m_y, -2m_y)$ (called *minimally*

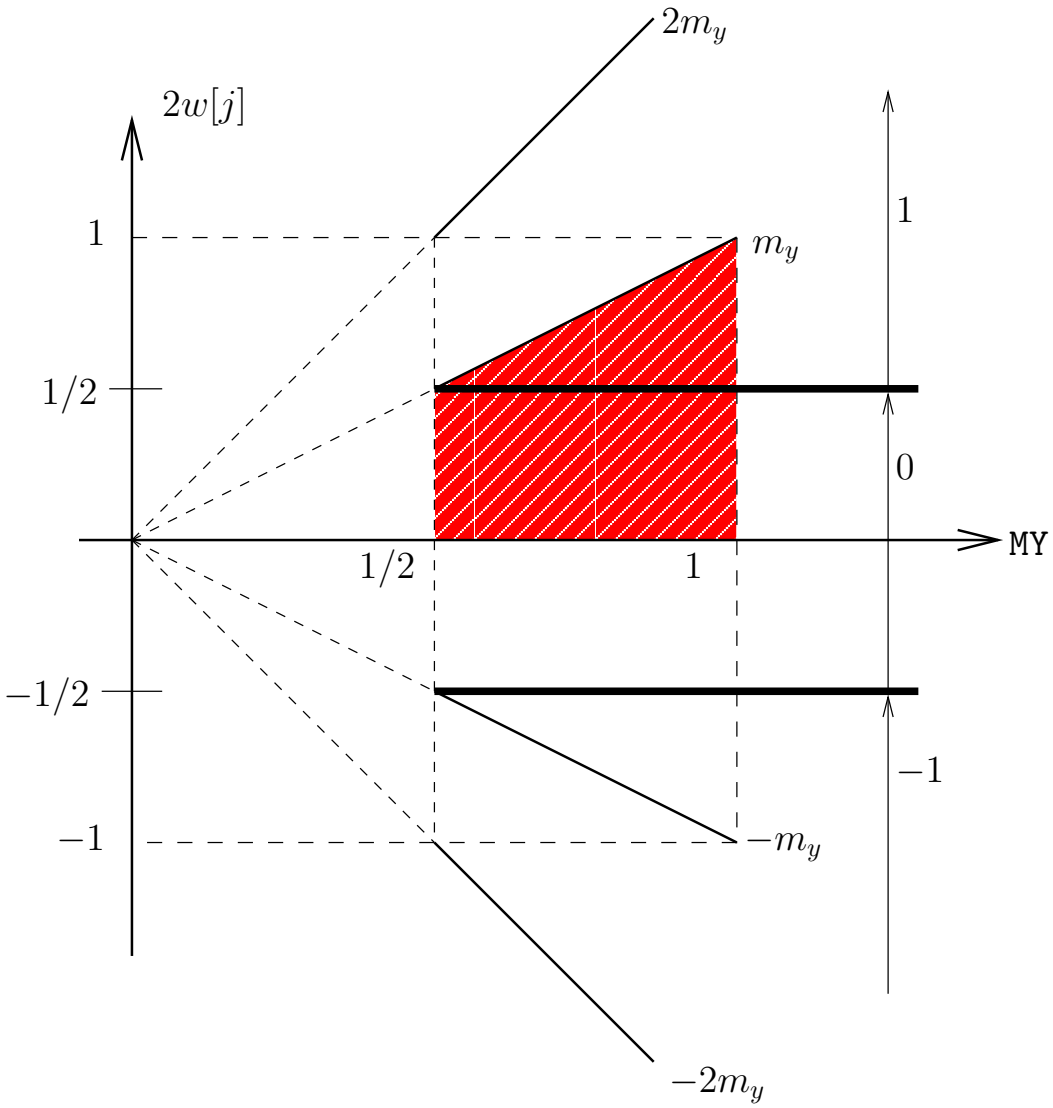


Figure 9.5: Radix-2 SRT division: P-D diagram

redundant) of the divisor have to be generated which is quite simple as they require simple shift. Whereas, when $a = 3$, two more multiples $(3m_y, -3m_y)$ (called *maximally redundant*) have to be generated which requires an additional operation, but at the same time, redundancy increases which simplifies the selection function.

We begin by the implementation of radix-4 division with quotient-digit set $\{-2, \ldots, 2\}$. The redundancy factor ρ can be computed from (9.4) as $\rho = 2/3$ and from (9.9) the bound for the residual is

$$|w[j]| \leq 2/3m_y. \tag{9.12}$$

So it is necessary to scale the first residual $w[0] = m_x$ to the proper range. Due to this scaling the first two leading bits of the quotient $Q = m_x/m_y$ might be zero and thus the quotient must be computed with two extra bits of precision. This computation requires an extra iteration. Figure 9.6 shows the positive half of the P-D diagram for the radix-4 SRT algorithm with quotient digit set $\{-2, \ldots, 2\}$. We can see that all the decision boundaries in the overlap region are not straight as was the case with radix-2 SRT algorithm. So, unlike for radix-2, the selection function for radix-4 does not solely depend on the constants but also on the divisor m_y. Since the selection

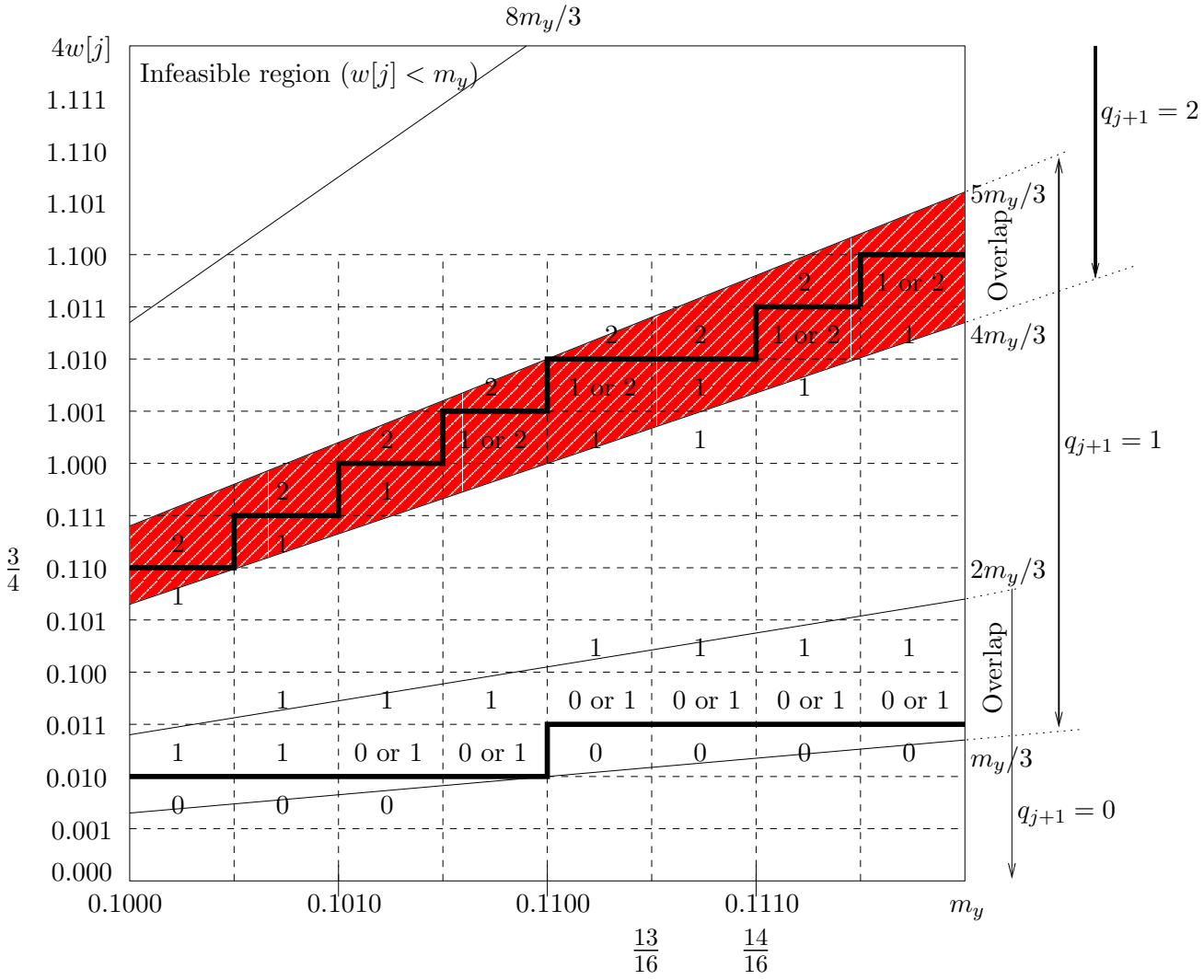


Figure 9.6: Radix-4 SRT division with quotient digit set {-2,...,2}: P-D diagram

function depends on a few bits of the divisor too, residuals are compared to the selection constants corresponding to each interval of the divisor.

In general, the selection function is implemented by first determining the interval of the divisor, and then a particular set of selection constants is chosen depending on this interval. This determination requires up to three comparisons in the case of the digit set $\{-2, \ldots, 2\}$ and at most one in the case of $\{-3, \ldots, 3\}$. It turns out that, for our target architecture, these comparisons increase the latency significantly and should be avoided as much as possible. Therefore we suggest below a comparison-free alternative for computing the two selection constants, say s_1 and s_2, corresponding to the digit set $\{-2, \ldots, 2\}$ [57]. Writing $\lfloor \alpha \rfloor$ for the usual floor function of a real number α and recalling that d has the form $m_y = 0.1y_2y_3y_4 \cdots y_n$ (since overlap is larger when $1/2 \leq m_y < 1$) with $y_i \in \{0, 1\}$, we derive empirically s_1 and s_2 and define them as the following functions of m_y:

$$s_1 = \frac{3}{4} + \frac{1}{2}y_2 + \frac{1}{4}y_3 + \frac{1}{8}y_4 - t,$$

where

$$t = \frac{1}{8}\left\lfloor y + \frac{3}{16} \right\rfloor,$$

and

$$s_2 = \frac{1}{4} + \frac{1}{8}y_2.$$

For a given m_y, the values s_1 and s_2 can thus be obtained without any comparison with m_y. Additionally, by definition of t, one has

$$t = \begin{cases} 0 & \text{if } m_y \in [\frac{1}{2}, \frac{13}{16}), \\ 1 & \text{if } m_y \in [\frac{13}{16}, 1), \end{cases}$$

and one may check that s_1 is exactly the "staircase" that belongs to the upper overlap region of the P-D diagram in Figure 9.6. Similarly, the "staircase" in the lower overlap region is defined by s_2.

We remark that the value of t actually depends only on $0.1y_2y_3y_4$. However, in our software context, getting these particular bits would require one more logical operation (mask). Computing t with the whole m_y is thus more efficient.

A radix-4 SRT implementation using the selection constants s_1 and s_2 is given in Figure 9.7. Although the maximally redundant set results in a simpler selection function, it does not provide faster implementation. We can see in Table 9.6, comparison of radix-4 SRT division with $\{-3, \ldots, 3\}$ and $\{-2, \ldots, 2\}$ digit sets, that the later results in a faster implementation. This is due to the fact that there are fewer divisor multiples.

9.4 Very-High Radix SRT Algorithm

Digit-recurrence algorithms benefit by increasing the radix since fewer iterations are needed but may become less efficient as the selection function may become more complex. The idea of the high radix SRT algorithm is to use a higher radix as well as to construct a simplified selection function [32, p. 103]. This simplified selection function is called *selection by rounding* [29]. It is based on selecting a fixed number of most significant bits of the *rounded* shifted residual as the quotient digit. The preliminary requirement of this selection function is to restrict the divisor to a range close to 1. This allows to construct the selection function independently of the divisor. Restricting the range of the divisor can be done by *prescaling the divisor* [91, 60, 31, 34]. Since the divisor is prescaled, the dividend must be prescaled too to preserve the value of the quotient as

$$\frac{prescale(m_x)}{prescale(m_y)} \approx Q$$

where $prescale(m_y) \approx 1$. After the prescaling step, multiple SRT iterations from (9.5) are performed. Each iteration consists of reduction of the residual by *multiplication* and *subtraction* and selection of the quotient digit by rounding. The final step is to obtain the correctly rounded quotient. It entails the computation of the remainder. The prescaling step highly influences the implementation of the algorithm. It determines the initial precision of the quotient and consequently, the radix of the algorithm. So, a detailed discussion about the prescaling step and the issues involved in its implementation shall be done in Section 9.4.1. The implementation of high radix iterations shall be presented in Section 9.4.2. We will see later that the high radix SRT algorithm can be well adapted for the target processor ST220. It does not mean that it can not be implemented for the processor ST231. But for ST231, a different algorithm is used since it provides a faster implementation. So during the discussion of the very-high radix algorithm we mainly focus on ST220.

```
1  w[0] ⟵ m_x
2  t ⟵ ⌊m_y + 3/16⌋ × 2^{-3}
     where m_y = 0.1 y_2 y_3 y_4 ··· y_n
3  s_1 ⟵ 3/4 + 2^{-1} y_2 + 2^{-2} y_3 + 2^{-3} y_4 − t
4  s_2 ⟵ 1/4 + y_2/8
5  n ⟵ ⌊n/2⌋ + 1
6  for j from 1 to n do
7      w[j] ⟵ 4 × w[j − 1]
8      if w[j] ≥ 0 then
9          if w[j] ≥ s_1 then
10             w[j] ⟵ w[j] − 2 × m_y
11             q_j ⟵ 2
12         else
13             if w[j] ≥ s_2 then
14                 w[j] ⟵ w[j] − m_y
15                 q_j ⟵ 1
16             else
17                 q_j ⟵ 0
18     else
19         if w[j] ≤ −s_1 then
20             w[j] ⟵ w[j] + 2 × m_y
21             q_j ⟵ −2
22         else
23             if w[j] ≤ −s_2 then
24                 w[j] ⟵ w[j] + m_y
25                 q_j ⟵ −1
26             else
27                 q_j ⟵ 0
```

Figure 9.7: Radix-4 SRT division algorithm with digit set {-2,. . . ,2}.

9.4.1 Prescaling

There exist different ways of performing the prescaling step, for example, scaling the divisor and the quotient. Three ways are given in [32, p. 104]. We employ the "prescaling of both the dividend and the divisor" way in our implementation. The divisor and the dividend are prescaled by the factor $\mathcal{F}$ so that the scaled divisor Z is

$$1 - 2^{-b} \leq Z = \mathcal{F} m_y \leq 1 + 2^{-b} \tag{9.13}$$

where b is the initial precision of the quotient and $r = 2^b$ is said to be the radix of the algorithm. Typically, $8 \leq b \leq 16$ in order to merit for the term "very-high radix". The initial residual is the scaled dividend $w[0] = \mathcal{F} m_x$. The scaling factor $\mathcal{F}$ is an approximation of the reciprocal of the divisor. There exist different methods such as look-up tables, linear interpolation [34], polynomial approximation, etc., to compute $\mathcal{F}$. Since the cost of a data miss is around 130 cycles, tables can not be used on the target architecture. We compute $\mathcal{F}$ through polynomial approximation. We find $p(m_y)$ in Maple 9.5 [55], a *minimax* polynomial approximation to $1/m_y$ on $[1, 2)$.

A degree one polynomial that can easily be evaluated, obtained in Maple 9.5, is

$$p(m_y) = -2^{-1} m_y + 1.457106781225\ldots.$$

Since m_y can always be stored already aligned (shifted), evaluation of this polynomial only requires an addition. To compute polynomials on the target architecture we have to take into account the following:

1. Representation of coefficients: the infinite binary expansion of coefficients is rounded to nearest after the mth fractional bit to be able to store it on a 32-bit register. This rounding induces the representation error which is less than 2^{-m-1}. The value of m depends on many parameters as described in Section 1.8.

2. Evaluation of intermediate steps to arrive at the final result: see Section 9.4.1.1 to know how polynomials are evaluated on the target architecture due to efficiency reasons. Also, see the examples given in Section 1.8 which show how intermediate steps are evaluated in order to have maximum accuracy possible. Since truncated multiplications are used in intermediate steps, this induces evaluation error and eventually a loss of accuracy.

Taking into account, both the representation and the evaluation error, after the computation of $Z = \mathcal{F}m_y$, we have $1 - 2^{-b} \leq Z \leq 1 + 2^{-b}$ with $b = 4$. Now, the first 4 bits of the quotient come from prescaling and multiple iterations in radix $r = 2^4 = 16$ can be performed to obtain the remaining quotient bits. From (9.11) six iterations must be performed. So, with the initial precision of 4 bits one can not implement a very-high radix algorithm.

In order to fully exploit the idea of high radix SRT algorithm we try higher radices to reduce, let us say to 1 or 2, the number of iterations. This obliges us to search a polynomial which can approximate the reciprocal of the divisor and gives the required initial precision. A polynomial of higher degree will be used to increase b. But, evaluation of intermediate steps of a polynomial may involve some scaling operations which might be expensive. Avoidance of these scaling operations leads to a compensation for the initial precision. So, the *cost* of the evaluation of a polynomial must be balanced with the *required initial precision* provided by it. In short, one must try to maximize the speed of and/or minimize the errors occurred in the implementation of the approximation. In our division implementation for ST220 radix $r = 512$ is used. Why this radix has been preferred for this particular processor will be explained later.

In Section 9.4.1.1 we present a method for evaluation of polynomials which is well suited for our target architecture. We give complete analysis of the polynomial evaluation and compare different polynomials with reference to the target architecture. In Section 9.4.1.2 we try different polynomials of higher degrees so as to find the best possible value of b without degrading much the latency involved in their evaluation.

9.4.1.1 Polynomial Evaluation

The common known method for evaluation of polynomials, Horner method, does not prove to be the optimal method concerning our target processors. This is due to the following reasons.

1. Horner method evaluates a polynomial by performing multiplication and addition in a sequential way.

2. The target architecture can perform two multiplications and four additions in parallel.

Although Horner method evaluates a polynomial with the minimum number of operations, it does not exploit parallelism. The method we propose proves to be more efficient than the Horner method as it adapts well to the target architecture. Although our method performs more operations (additions and multiplications) in comparison to the Horner method, they are performed in parallel and so the evaluation is faster.

Horner method evaluates a degree-3 polynomial in 12 cycles as

$$((u_3 m_y + u_2)m_y + u_1)m_y + u_0.$$

Our method, which is close to Estrin's method ([59, p. 488], [68]), evaluates the same polynomial in 8 cycles as

$$(u_3 m_y + u_2)\ m_y^2 + u_1 m_y + u_0.$$

Moreover, if some of the coefficients are a power of 2 then the number of cycles might decrease. Our method performs one more multiplication than the Horner method but the number of additions is the same, in case of a polynomial of degree 3. The efficiency of this method comes from the fact that operations are performed in a particular order and in a parallel fashion. Note that for degree-3 polynomials the approach of our method is the same as the approach of Estrin's method but as we go for higher degree polynomials we find that our method is more adapted to the target architecture. The order of evaluating a polynomial of degree 5 in a parallel fashion on the target architecture is given in Figure 9.8. The evaluation using our method takes 10 cycles whereas the one based on the Horner method would take $5(3+1) = 20$ cycles. The approach of evaluating the degree-5 polynomial is

$$v(m_y) = \underbrace{(\underbrace{v_5 m_y + v_4}_{T_1})m_y^4 + \underbrace{(\underbrace{v_3 m_y + v_2}_{T_2})m_y^2 + \underbrace{v_1 m_y + v_0}_{T_3}}_{T_4}}$$

and the coefficients are given in Table 9.8 below.

Cycles	1	2	3	4	5	6	7	8	9	10
MUL1	$v_5 m_y$	$v_3 m_y$	–	–	–	$T_2 x^2$	–	–	–	–
MUL2	x^2	$v_1 m_y$	–	x^4	–	–	$T_1 x^4$	–	–	–
ADD1	–	–	–	T_1	T_2	–	–	–	T_4	–
ADD2	–	–	–	–	T_3	–	–	–	–	$T_1 x^4 + T_4$

Figure 9.8: Evaluation of a polynomial of degree 5 on the target architecture.

We also tried evaluation of higher degree polynomials. A comparison of our method with the Horner method in terms of the number of cycles required for evaluation is given in Table 9.2. We assume in Table 9.2 that none of the coefficients of the polynomial of any degree is a power of 2, that is, none of the multiplications can be replaced by simple *shifts*. We infer that, in general, for the target architecture, using our method of evaluation, a polynomial of odd degree is better than a polynomial of even degree.

After discussing how to improve the latency involved in the evaluation of polynomials, we now discuss how to minimize the errors occurred during evaluation. Since the multipliers available are not the same on both the target processors, it may be possible that some polynomials are only suitable for the processor ST231 which has a full-width multiplier.

We have evaluated polynomials of different degrees on ST220 and ST231, and the initial precision obtained can be seen in Figure 9.9. The difference in the initial precision is due to increased evaluation error. In case of ST220, since all the coefficients are represented on 16 bits and truncated multiplications are used in intermediate computations (computation of m_y^2 is performed as $m_{y32} \times m_{y16} \rightarrow m_{y32}^2$), gain of using a polynomial of higher degree is offset by a significant loss in accuracy of intermediate results. We conclude from Figure 9.9 that evaluation of polynomials of

Polynomial degree	Number of cycles	
	Estrin-like method	Horner method
2	7	8
3	8	12
4	10	16
5	10	20
6	11	24
7	11	28

Table 9.2: Comparison of evaluation time of a polynomial for two different methods.

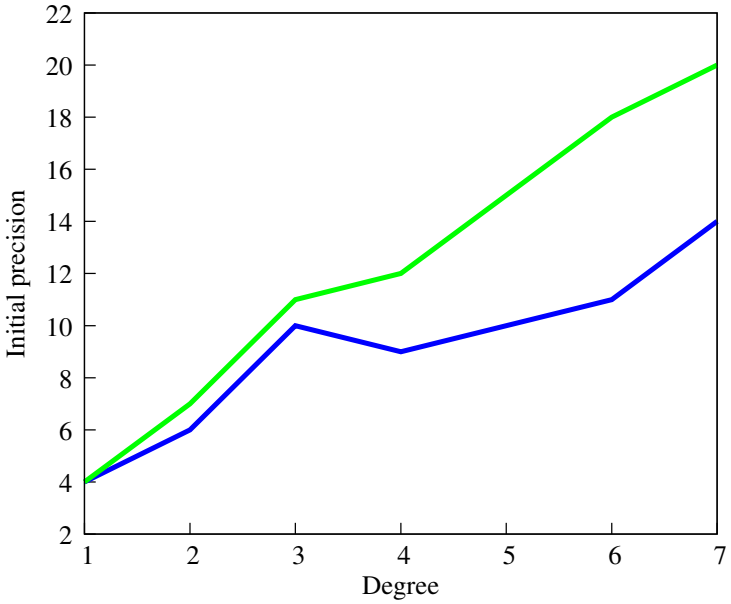


Figure 9.9: Polynomials of different degrees and the precision obtained after their evaluation on ST220 and ST231.

higher degree on ST220 should be avoided as it increases the computations without producing the approximation of desired precision.

The implementation of the approximation used in our floating-point division implementation has been formally proved through a tool called Gappa [65]. Gappa is a tool which takes into account all the errors occurred at each computation step and formally proves that after performing all these operations, the final error lies in a particular interval. See Appendix A to know how Gappa has helped us in verifying that the evaluation of a particular polynomial produces an approximation with the desired precision.

9.4.1.2 Polynomial Approximation

In this section we try different polynomials of higher degrees so as to find the best possible value of b, the initial precision of the quotient, in order to realize and implement a *very-high radix* SRT algorithm.

A polynomial of degree 2 gives $b = 6$ and four high radix iterations must be performed to obtain the remaining bits of the quotient Q. In order to have $8 \leq b \leq 16$ we try a polynomial of degree 3 that gives $b = 8$ and thus three SRT iterations are required. Increasing b slightly might

be interesting since increment of b from 8 to 9 permits to get rid of *one complete iteration*. So a polynomial of higher degree should be tried. But there exist a tradeoff between a polynomial of higher degree (more computation) followed by fewer iterations and a polynomial of lower degree (less computation) followed by more iterations. The former choice recommends a polynomial of degree 4 or 5 but as said before, these polynomials are not worth evaluating on ST220 as the initial precision is not obtained as expected. So we try to obtain $b = 9$ by using two degree-3 polynomials.

We propose to find a minimax polynomial approximation to $1/m_y$ on two different intervals $[1, 3/2)$ and $[3/2, 2)$ instead on a single interval $[1, 2)$ as before [57]. Two different polynomials for each interval are proposed. The degree-3 polynomial is

$$p(m_y) = (-p_3 m_y + p_2)\ m_y^2 - p_1 m_y + p_0 \tag{9.14}$$

and the coefficients of each polynomial are given in Table 9.3.

$m_y \in$	Coeffs	Decimal values
$[1, 3/2)$	p_0	$3.2828333\ldots$
	p_1	$4.0136915\ldots$
	p_2	$2.1661051\ldots$
	p_3	$0.4354185\ldots$
$[3/2, 2)$	p_0	$2.3153857\ldots$
	p_1	$2.0034469\ldots$
	p_2	$0.7678062\ldots$
	p_3	$0.1099684\ldots$

Table 9.3: Coefficients of each polynomial in (9.14).

Splitting the interval into two halves helps to produce more precise approximations. Now we have relatively lower-degree polynomials and yet the desired value of $b = 9$. Comparison of m_y to $3/2$ can be done in parallel to the evaluation. Now, floating-point division can be implemented through this polynomial for the prescaling step followed by two SRT iterations and finally the correct rounding of the quotient. This implementation of division is already twice faster (at least) than the other implementations studied before. But if the overall speed of the division has to be minimized further then a fast implementation of each of the three steps is necessary. Here we consider some optimizations done on the evaluation of the polynomial step.

We see from Figure 9.8 that only the coefficients of odd powers of m_y are multiplied. For example, in case of a degree-3 polynomial, $p_3 m_y$ and $p_1 m_y$ are crucial multiplications. These multiplications can be replaced by simple shifts if p_3 and p_1 are powers of two (we assume that $m_y \neq 1$). We have found another polynomial which gives an initial precision of $b = 9$ and has special coefficients which are power(s) of 2. Multiplications with these coefficients can be replaced by simple shifts and adds which reduces the latency involved. This optimization contributes to the overall speed of the division.

The proposed polynomial approximates $1/(1 + f_y)$ where $f_y \in [0, 1)$. This is equivalent to a polynomial which approximates $1/m_y$ where $m_y \in [1, 2)$ since $m_y = 1 + f_y$. Instead of splitting into two halves, the intervals are splitted in an unconventional manner, $[0, 0.404)$ and $[0.404, 1)$. So, two different polynomials of degree 3 are proposed as

$$u(f_y) = (-u_3 f_y + u_2)\ f_y^2 - u_1 f_y + u_0 \tag{9.15}$$

and the coefficients for two intervals are given in Table 9.4 below.

$f_y \in$	Coeffs	Decimal	Hexadecimal	Binary
$[0, 0.404)$	u_0	$0.99991279\ldots$	0x3FFE923E	00 . 111111 . . .
	u_1	0.9921875	0x7F000000	0 . 1111111 . . .
	u_2	$0.89556096\ldots$	0x3950DEF2	00 . 111001 . . .
	u_3	0.5	0x4000000	0 . 1000000 . . .
$[0.404, 1)$	u_0	$0.97599094\ldots$	0x3E76A2BD	00 . 111110 . . .
	u_1	$0.82339477\ldots$	0x69650000	0 . 1101001 . . .
	u_2	$0.47217888\ldots$	0x3C70588E	0 . 0111100 . . .
	u_3	0.125	0x20000000	. 00100000 . . .

Table 9.4: Coefficients of the polynomial in (9.15). Hex representation is used in the implementation.

For the first interval, $u_3 f_y$ is a simple shift `f_y >> 1` and $u_1 f_y$ is a simple subtraction `f_y - (f_y >> 7)` (the quantity f_y shifted right by 7 bits can be prestored in a register). For the second interval, unfortunately $u_1 f_y$ is an explicit multiplication but $u_3 f_y$ is a simple shift. Moreover, the evaluation time for both polynomials is the same, that is, 7 cycles which is *one cycle less* than taken by usual polynomials. Note that the hex values given in Table 9.4 are stored in a 32-bit register and are used in the implementation. This kind of representation which has leading zeros in the corresponding binary expansion allows to avoid some scaling operations during evaluation. This approximation of the reciprocal helps to obtain first 9 bits of the quotient after the prescaling step. So we perform two high radix iterations to have the complete quotient as described in Section 9.4.2. Note that after 9, the next interesting value of b is 13 as only one high radix iteration will be required (as before, we aim to obtain a few extra bits after a 24-bit quotient to be able to perform rounding). This polynomial with some optimizations done to avoid some scaling operations during its evaluation contributes to the overall latency of division by reducing it by three cycles.

9.4.2 High Radix Iteration

We begin with the implementation of high radix SRT iterations once the original dividend and divisor are prescaled to $w[0] = \mathcal{F}m_x$ and $Z = \mathcal{F}m_y$ respectively. After prescaling we have $1-2^{-9} \leq Z \leq 1+2^{-9}$ and the radix $r = 512$. We use the quotient-digit set $\{-511, \ldots, 511\}$. The redundancy factor ρ can be computed from (9.4) as $\rho = 1$ and from (9.9) the bound for the residual can be given as

$$|w[j]| \leq m_y. \tag{9.16}$$

So it is necessary to scale the first residual $w[0] = \mathcal{F}m_x$ to the proper range. Due to this scaling the first two leading bits of the quotient $Q = m_x/m_y$ might be zero and thus the quotient must be computed with two extra bits of precision. Also, because of selection by rounding, q_1 can be 512 but in this case the next residual $w[1]$ will be negative and thus q_2 will be negative too, which will offset q_1 to produce $Q < 1$. So, an extra guard bit is maintained to store the carry, if generated by rounding the residual.

The first quotient digit is obtained by rounding the shifted residual (prescaled dividend) as

$$q_1 = \left\lfloor 512w[0] + \frac{1}{2} \right\rfloor.$$

The SRT iteration in (9.5) is modified as

$$w[0] = \mathcal{F}m_x, \tag{9.17a}$$

$$w[j] = rw[j-1] - Z \times q_j, \quad j = 1, 2, \tag{9.17b}$$

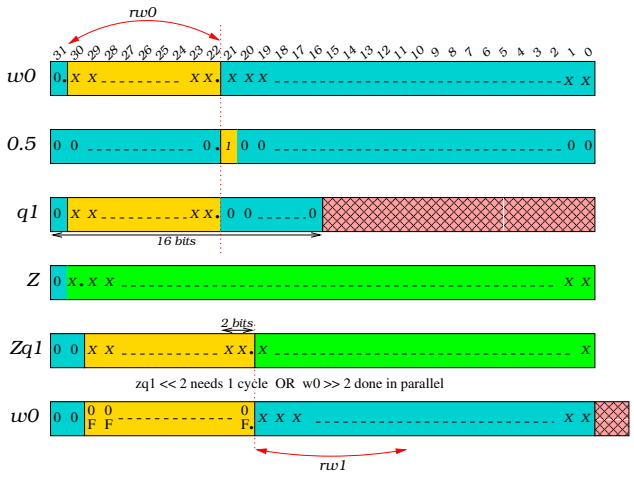
and two iterations are performed as shown in the pseudo-code in Figure 9.10.

```
1  // Prescaling
2  Z ← Fm_y
3  w[0] ← Fm_x
4  // First quotient digit
5  w[0] ← 512w[0]
6  q_1 ← ⌊w[0] + 1/2⌋
7  // SRT iteration
8  w[1] ← w[0] − q_1 × Z
9  w[1] ← 512w[1]
10 q_2 ← ⌊w[1] + 1/2⌋
11 w[2] ← w[1] − q_2 × Z
12 w[2] ← 512w[2]
13 q_3 ← ⌊w[2] + 1/2⌋
14 q ← q_1 2^18 + q_2 2^9 + q_3
15 // Correct rounding (compute remainder)
16 Rem ← m_x − q × m_y
17 if Rem ≤ 0 then
18     q ← q − 1
19 // Rounding (nearest even)
20 Q ← ROUND(q)
```

Figure 9.10: Very-high radix SRT division algorithm with radix $r = 512$ and digit set {-512,...,512}.

In Figure 9.11 we provide a piece of C code where one high radix iteration is performed and the quotient digit is selected by rounding the shifted residual. It shows how the residual, the quotient, etc. are stored on a 32-bit register. The term $rw[0]$ is not computed explicitly. Instead of shifting $w[0]$, one can imagine that the decimal point is shifted right by 9 bits ($r = 2^9$). The quotient digit q_1 is computed by first adding 0.5 to $rw[0]$ and then masking to have the integral part $\left(\left\lfloor rw[0] + \frac{1}{2} \right\rfloor\right)$. An extra guard bit on the extreme left is required to avoid overflow. One of the main reason of choosing high radix SRT algorithm for ST220 is that since q_1 can easily be represented on a 16-bit register, the available multiplier can take 32-bit Z and 16-bit q_1 and can return a 32-bit product. The intrinsic function `__st220mulhhs` returns the significant 32 bits of a 48-bit product. The product will have two leading zeroes. Now, $rw[0] - Z \times q_1$ can not be performed as the binary point is not aligned. So instead of shifting Zq_1 left by two bits which would require an extra cycle, $w[0]$ is shifted left, which can be done in parallel with the multiplication. This kind of optimization saves one cycle in each iteration. Consequently, for the second iteration $rw[1]$ is computed by shifting $w[1]$ left by eleven bits instead of nine bits.

After two iterations the quotient q is formed and in order to correct the last bit the remainder is computed as $Rem = m_x - q \times m_y$ using the original dividend and the divisor and *not* their prescaled counterparts. Since q has 24 bits and one guard bit, we have to perform a multiplication of the form 25×24. On ST220, this rather long multiplication is done by performing four $16 \times 32 \rightarrow 32$ multiplications on smaller arguments after splitting. This correct rounding seems to be twice expensive on ST220 than on ST231.




```
q1=(w0 + 0x00200000)
     & 0xFFC00000;

w1=(w0 >> 2)
     - __st220mulhhs(Z,q1);

w1=w1 << 11;

q2=(w1 + 0x00200000)
     & 0xFFC00000;
```

Figure 9.11: Machine interpretation of and optimization done on the high radix iteration.

The speed achieved with this implementation based on the high radix SRT algorithm is shown in Table 9.6. It turns out to be faster than all other implementations presented before. In particular, it is more than twice faster than the implementation of the basic restoring algorithm and almost three times faster than the division available from the native STMicroelectronics library.

9.5 Functional Iteration Algorithms

In comparison to the digit recurrence algorithms, these algorithms become more attractive when an enhanced multiplier is available. Multiplication is the fundamental operation in these *quadratically convergent* algorithms. This means that instead of retiring a fixed number of quotient bits, they double it in each iteration. So the number of iterations has been reduced but the latency of each iteration has increased now. So it is important to know and to understand whether the availability of a fast and accurate multiplier can justify to consider this tradeoff between the two classes of division algorithms.

In general, these algorithms refine an already computed initial approximation through a converging algorithm. This section presents implementations based on two such algorithms, that is, Newton-Raphson in Section 9.5.1 and Goldschmidt in Section 9.5.2.

Division can be viewed as a *product* of the dividend m_x and the reciprocal of the divisor m_y, that is,

$$Q = m_x/m_y = m_x \times (1/m_y).$$

Here, an efficient approximation of the reciprocal helps to reduce the number of iterations of the converging algorithm. Newton-Raphson tries to refine the reciprocal up to desired precision before

multiplying it with the dividend. On the other hand, Goldschmidt first multiplies the initial approximation with the dividend and then improves it up to the desired precision. We will see later that since Newton-Raphson involves dependent multiplications, it proves to be slower than Goldschmidt when compared on the target processor. As said before, approximation through look-up tables is expensive on the target architecture. So we try to approximate the reciprocal through a minimax polynomial approximation and implement it by performing some additions and multiplications. Since the implementation of iterative algorithms is considered for ST231 because of the availability of the full-width multiplier, a polynomial of higher degree can easily be implemented.

The major drawback for these algorithms is the computation of a *correctly rounded* quotient. Unlike digit recurrence algorithms, the remainder is not produced after the last iteration, instead it must be computed explicitly. So, correct rounding requires either computation of extra quotient digits (more iterations), or computation of all intermediate steps in larger precision [69] unless the iterations are performed using a fused multiply-and-add operation [22]. For more details on obtaining a correctly rounded quotient see [22, 56, 63, 72].

9.5.1 Newton-Raphson Algorithm

The Newton-Raphson algorithm [41] consists of finding an approximation ($\mathcal{F} \approx \frac{1}{m_y}$) to the reciprocal using the iteration

$$\mathcal{F}_{i+1} = \mathcal{F}_i(2 - \mathcal{F}_i m_y)$$

and the error in the approximation decreases quadratically. This algorithm allows to double the number of quotient bits at each iteration. Initial reciprocal approximation is obtained through two different polynomials of degree 5 which are

$$v(m_y) = (-v_5 m_y + v_4)\ m_y^4 + (-v_3 m_y + v_2)\ {m_y}^2 - v_1 m_y + v_0 \tag{9.18}$$

where the coefficients for two intervals are given in Table 9.5

$m_y \in$	Coeffs	Decimal	Hexadecimal	Binary
$[1, 1.5)$	v_0	4.91581623...	0x13A9CBBC	000100 . 11...
	v_1	10.02742341...	0x503829C5	01010 . 0000...
	v_2	10.86405751...	0xADD32DF9	1010 . 1101...
	v_3	6.59377369	0xD30031B3	110 . 10011...
	v_4	2.12570681	0x880B9496	10 . 001000...
	v_5	0.28438519	0x2466BBF6	0 . 0100100...
$[1.5, 2)$	v_0	3.47008467...	0x1BC2BBC2	00011 . 011...
	v_1	5.00689970...	0x501C42DC	0101 . 0000...
	v_2	3.84500280...	0x7B0A4354	011 . 11011...
	v_3	1.65748198	0x6A142F53	01 . 101010...
	v_4	0.38028095	0x30AD0BD9	0 . 0110000...
	v_5	0.03627925	0x094998D5	. 00001001...

Table 9.5: Coefficients of the polynomial in (9.18). Hex representation is used in the implementation.

The polynomial is evaluated on ST231. The polynomial approximation provides first 14 bits of the reciprocal and one Newton-Raphson iteration doubles it. The final quotient is obtained

by multiplying the computed reciprocal with the dividend. Finally, the remainder is computed in order to correct the last bit of the quotient. The pseudo-code of the algorithm is given in Figure 9.12.

```
1  F1 ← v(my)
2  F2 ← F1(2 − F1my)
3  q ← F2mx
4  // Correct rounding (compute remainder)
5  Rem ← mx − q × my
6  if Rem ≤ 0 then
7      q ← q − 1
8  // Rounding (nearest even)
9  Q ← ROUND(q)
```

Figure 9.12: Newton-Raphson division algorithm with an initial reciprocal approximation computed through a degree-5 polynomial.

The speed achieved with this implementation based on the Newton-Raphson algorithm is shown in Table 9.6. Indeed it is faster than all the other implementations including the one based on very-high radix SRT algorithm but the implementation and the optimizations done are only relevant for ST231. See Section 9.6 for more information on the performances of this implementation.

9.5.2 Goldschmidt Algorithm

The Goldschmidt algorithm [44, 41, 30, 74, 64] for computing m_x/m_y can be given as follows. Initial approximation of $1/m_y$ is computed as $\mathcal{F}_1$. Then multiplying the dividend and the divisor by this approximation gives

$$\frac{m_x}{m_y} = \frac{m_x\mathcal{F}_1}{m_y\mathcal{F}_1} = \frac{q_1}{Z_1}.$$

The goal of Goldschmidt algorithm consists in finding a sequence $\mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_3, \ldots$ by which to multiply the dividend and the divisor such that it drives the product $Z_i = m_y\mathcal{F}_1\mathcal{F}_2 \ldots \mathcal{F}_i$ towards unity and consequently, the product $q_i = m_x\mathcal{F}_1\mathcal{F}_2 \ldots \mathcal{F}_i$ towards the quotient as i approaches infinity. Hence, continuing in this manner yields

$$\frac{m_x}{m_y} = \frac{m_x\mathcal{F}_1\mathcal{F}_2 \ldots \mathcal{F}_i}{m_y\mathcal{F}_1\mathcal{F}_2 \ldots \mathcal{F}_i} = \frac{q}{1} \quad (i \to \infty).$$

In our implementation of the above algorithm, initial reciprocal approximation is obtained through a polynomial of degree 5 (same as in the Newton-Raphson method in Section 9.5.1) which gives $\mathcal{F}_1 = 1/m_y$ such that

$$1 - 2^{-14} \leq Z_1 = m_y\mathcal{F}_1 \leq 1 + 2^{-14}.$$

The initial precision of 14 bits (first quotient digit) suffices to perform only one iteration of Goldschmidt algorithm as shown in the pseudo-code in Figure 9.13.

The terms $m_x \times \mathcal{F}_1$ and $q_1 \times \mathcal{F}_2$ must be computed using a $32 \times 32 \to 32$ multiplier so this algorithm is not suitable for performing division on ST220.

```
1  F1 ← v(my)
2  Z1 ← my × F1
3  q1 ← mx × F1
4  F2 ← 2 − Z1
5  q ← q1 × F2
6  // Correct rounding (compute remainder)
7  Rem ← mx − q × my
8  if Rem ≤ 0 then
9      q ← q − 1
10 // Rounding (nearest even)
11 Q ← ROUND(q)
```

Figure 9.13: Goldschmidt algorithm with an initial reciprocal approximation computed through a degree-5 polynomial.

9.6 Performances

All the performances have been measured both on ST220 and ST231 processors. Table 9.6 reports the performances of different floating-point division implementation based on various division algorithms. The division implementation in the original algorithm performs an integer division through the available instruction `divs` on ST220, and 64-bit multiplication and addition which are emulated and are expensive. Due to this, even the basic division algorithms (restoring and nonrestoring) which perform 25 iterations inside a loop are faster than the original division. Theoretically, the radix-2 SRT algorithm is faster than the nonrestoring one since it employs redundancy but practically it is not true in the case of the target architecture. The radix-2 SRT algorithm is slightly slower than nonrestoring algorithm.

Between the two digit sets of the radix-4 SRT algorithm, the general belief is to choose the bigger digit set as the redundancy factor ρ is higher in this case and it is responsible for the amount of overlap region too. In short, it is an important factor when implementing division. But the performances measured on the target processors for both digit sets dispute the general belief. It is due to the fact that there are fewer divisor multiples to deal with in the case of the smaller digit set.

Aiming to further accelerate the division, we tried some different algorithms and optimized them to exploit the parallelism available in the target processors, like performing two multiplications and four additions or logical operations in parallel. Through multiplications these algorithms computed the result very fast but on the other hand they posed the problem of accuracy and error control. Due to these problems different algorithms have been chosen for two different target processors. Finally we propose very-high radix SRT algorithm for ST220 and Goldschmidt algorithm for ST231. For both processors a speed-up factor of at least three has been achieved in comparison to the implementation of the division in the original library.

Division methods	Timings (# cycles)	
	ST220	ST231
Original	179	177
Restoring	134	134
Nonrestoring	117	117
Radix-2 SRT	131	131
Radix-4 SRT with $\{-2, \ldots, 2\}$	125	125
Radix-4 SRT with $\{-3, \ldots, 3\}$	155	155
Very-high radix SRT	60	n.a.
Newton-Raphson	n.a.	52
Goldschmidt	n.a.	48

Table 9.6: Performances of the different division methods on ST220 and ST231.

Chapter 10

Reciprocal

Many modern processors like Itanium and PowerPC have a reciprocal estimate instruction. They perform division by iteratively refining this estimate to achieve the desired accuracy. In many graphics applications too reciprocal is frequently used and through a reuse of common divisors higher performance can be achieved. Although the core algorithmic part is exactly the same as in division except a saving of one floating-point multiplication, other optimizations can be done to simplify the handling of special values, the trivial computation step and the correct rounding of the result to justify support of this operation in FLIP. These optimizations influence the overall latency of this operation. Concerning FLIP, a speed-up factor of 1.2 has been achieved when computing the reciprocal of y instead of performing the division $1/y$. In this chapter we have assume an acquaintance with the implementation issues for the target processors discussed in the division chapter. We focus on what is different from division and look for further optimizing it.

10.1 Floating-Point Reciprocal

The implementation of the floating-point reciprocal is almost the same as of the floating-point division. The only basic difference, between their implementation, is that one floating-point multiplication (emulated as integer) is saved. So, we will only discuss about this difference and will check whether further optimizations can be done.

The input for floating-point reciprocal is a single precision floating-point operand y represented by (s_y, e_y, f_y). The implementation of this operation supports no subnormals and only round-to-nearest-even mode. The output is a single precision floating-point number $z = \circ(1/y)$. The mantissa m_y is constructed (Section 1.8) and a short description of floating-point reciprocal, which is very close to floating-point division, can be given as follows.

1. If y is $\pm\infty$, ± 0 or NaN, return the result as defined in Table 10.1.
2. Reciprocal of the mantissa $R = 1/m_y$ is computed.
3. The intermediate result exponent is computed as
$$e_z = 254 - e_y.$$
4. The sign of the result is $s_z = s_y$.
5. If $m_y \neq 1$, always normalize and update the exponent.

6. Round the exact reciprocal R.

7. Determine exceptions by verifying the exponent of the result.

10.2 Algorithm and Implementation

Since reciprocal can be treated as a division when the dividend is always one, we will not present the algorithmic part. We will only discuss how to implement each step of the basic description as well as further optimizations that can be done in this case. Optimized handling of special values is done in the same way as for other basic operations. To obtain the reciprocal of the mantissas, only the fast algorithms for division have been implemented. We have not considered the slower algorithms like nonrestoring or radix-2 SRT. The implementation of the floating-point reciprocal for both the processors is based on the same algorithms as those discussed for floating-point division.

10.3 Special Values

In case y is a special value the result can be chosen from the Table 10.1.

y	NaN	$+\infty$	$-\infty$	$+0$	-0
z	NaN	$+0$	-0	$+\infty$	$-\infty$

Table 10.1: All the possible predefined values of the reciprocal z when y is a special value.

10.4 Reciprocal of Mantissa

Among the three main classes of division algorithms, the implementation based on digit recurrence algorithms turns out to be slower than the implementations based on other classes. So, for the same reasons as described in the division chapter, we consider very-high radix algorithm for ST220 and iterative algorithms for ST231. We now explain for each algorithm the modifications that should be done on the division implementation.

10.4.1 Very-High Radix SRT Algorithm

The prescaling step has been implemented using the same degree-3 polynomial from (9.15) which produces the initial precision $b = 9$ and so the scaled divisor Z is such that

$$1 - 2^{-9} \leq Z = \mathcal{F}m_y \leq 1 + 2^{-9}.$$

The initial residual is the same as the approximation, that is, $w[0] = \mathcal{F}$. Once the initial residual and the scaled divisor have been obtained, two SRT iterations follow with a quotient digit selection function by rounding the residual after each iteration. The first 9 bits of the reciprocal come from the prescaling step and the remaining bits are obtained after the SRT iterations. In case of reciprocal, one saves the multiplication involved in prescaling the dividend.

10.4.2 Newton-Raphson Algorithm

We know that the Newton-Raphson algorithm consists of finding an approximation of the reciprocal. A 14-bit approximation is obtained from a degree-5 polynomial as in the case of division. A single Newton-Raphson iteration will produce a reciprocal twice as precise as before. Finally, this reciprocal is correctly rounded. Here again, one multiplication is saved.

10.4.3 Goldschmidt Algorithm

The same degree-5 polynomial as for the Newton-Raphson algorithm has been used to obtain the initial approximation of $1/m_y$ and it gives an initial precision of 14. Therefore, $1 - 2^{-14} \leq Z_1 = m_y\mathcal{F}_1 \leq 1 + 2^{-14}$. One Goldschmidt iteration suffices to find the complete reciprocal. The algorithm can be given as

```
1  F1 ← v(my)
2  Z1 ← my × F1
3  F2 ← 2 − Z1
4  q ← F1 × F2
5  // Correct rounding (compute remainder)
6  if q × my ≥ 1 then
7      q ← q − 1
8  // Rounding (nearest even)
9  R ← ROUND(q).
```

10.5 Rounding

For all three algorithms, correct rounding of the reciprocal is performed by only testing if $q \times m_y \geq 1$. The product here is a multiplication of the type 25×24 which can be performed faster on ST231 than on ST220.

10.6 Performances

The performances of our implementation of floating-point reciprocal based on these different algorithms measured both on ST220 and ST231 processors are given in Table 10.2. Note that the original library supports this operation by performing floating-point division. That is why the timings here are the same as for the division. All the three implementations have been inspired from the implementation of the floating-point division. We can see that there is an improvement of nine cycles for each implementation in comparison to division. This is due to a simplified handling of special values and a gain of one multiplication. But more importantly, because of single input operand the trivial computation step is less complex and has been implemented through a conditional select instruction (see Section 2.2.4). For division, this step requires a costly jump.

Different algorithms	Timings (# cycles)	
	ST220	ST231
Original	179	177
High-radix SRT	51	n.a.
Newton-Raphson	n.a.	43
Goldschmidt	n.a.	39

Table 10.2: Performances of the implementations based on different algorithms on ST220 and ST231.

Chapter 11

Square Root

This is the last operation in the set of basic arithmetic operations specified by the IEEE standard. Concerning complexity, square root can be seen as equivalent to division. It is a basic and essential operation in many scientific and engineering applications. Some typical applications include: solution of quadratic equations, trigonometry, error computation, statistics, graphics, etc. Most of the hardware units and software libraries compatible with the IEEE standard provide support for this operation. Most algorithms presented in the division chapter can be modified to implement square root. For instance, the basic principles of the nonrestoring or the Goldschmidt algorithm developed for division apply to square root. This chapter does not outline these algorithms which were already presented; rather it directly provides the implementations built up on these algorithms adapted to the target processors. The slower implementation of this operation in FLIP is almost the same as in the original library. But the fast implementation based on the Goldschmidt algorithm achieves a speed up factor of 2.5.

11.1 IEEE Floating-Point Square Root

The inputs for square root are

1. A single precision floating-point operand x represented by (s_x, e_x, f_x).
2. One of the four rounding modes to specify how to round the exact result.

The output is a floating-point number $z = \diamond(\sqrt{x})$, where $\diamond$ denotes the active rounding mode. The mantissa m_x is formed (Section 1.8) and the basic description of floating-point square root can be given as follows.

1. If x is $\pm\infty$, ± 0 or NaN, return the result as defined in Table 11.1.
2. If the unbiased exponent $e_x - 127$ is odd, a new mantissa is formed as $m_x \leftarrow 2m_x$. This allows to have an *integer* result exponent.
3. Obtain the square root as $T = \sqrt{m_x}$.
4. The intermediate result exponent is computed as

$$e_z = \lfloor (e_x + B)/2 \rfloor.$$

5. The sign of the result is $s_z = 0$.

6. No normalization is required since we consider $m_x \in [1, 4)$ which will produce a square root in the interval $[1, 2)$.

7. Round T according to the rounding mode. A post-normalization step might be required.

8. Neither underflow nor overflow can occur since $\sqrt{m_x} \in [1, 2)$ and $\sqrt{2^{e_x}} \in [1, 127]$.

11.2 Algorithm and Implementation

Here we discuss how to implement each step of the basic description. Unlike division, square root has been implemented using the nonrestoring and Goldschmidt algorithms only. Handling of special values has been very simplified. As for the division based on the Goldschmidt algorithm, the remainder must be computed in order to correct the last bit. Computation of the remainder requires a multiplication of type 32×32. Moreover, when the unbiased exponent is odd, since m_x falls into a different binade ($m_x \in [2, 4)$) now, the final square root will be obtained as $T = \sqrt{2}\sqrt{m_x}$. So, one more multiplication of the 32×32 is required. The hexadecimal constant 0xB504F333 ($\approx 1.4142135\ldots$) has been obtained after rounding to nearest the infinitely precise binary expansion of $\sqrt{2}$.

11.2.1 Special Values

In the case where x is a special value the result can be chosen from Table 11.1 below.

x	NaN	$+\infty$	$-\infty$	$+0$	-0	$x < 0$
z	NaN	$+\infty$	NaN	$+0$	-0	NaN

Table 11.1: All possible predefined values of the square root $\diamond(\sqrt{x})$ when x is a special value.

Handling of special values in FLIP is done as a two-step process:

1. The IEEE standard says that the square root of a negative number must be returned as NaN except for $\sqrt{-0}$, in which case it is -0. So, all $x < 0$ are changed into a NaN as shown by the pseudo-code on the right in Figure 11.1.

2. After this step, if e_x is either 255 or 0 then return x.

In Figure 11.1, two pseudo-codes are compared, the one in the original library on the left and the one in FLIP on the right. The signed integer X corresponds to the binary representation of x (see Section 1.4). Refer to Section 5.5.2.1 for details on the precomputation of $e_{x_{spl}}$.

How a negative number is changed into a NaN can be explained as follows. Let the binary string represented by X as $X[31:0] = [10\ldots00]$ which is an integer representation corresponding to -0. Since X is signed, $X >> 31$ will produce the string

$$X_{op1}[31:0] = [11\ldots1]$$

and $((0x80000000 - X) >> 31)$ will produce

$$X_{op2}[31:0] = [00\ldots0].$$

if $x = -0$ **then return** NaN;
if $s_x < 0$ **then return** NaN;
if $x = 0$ **then return** 0;
if $e_x = 255$ **then return** x;

$X = X\text{OR } ((X >> 31)\text{AND } ((0x80000000 - X) >> 31))$;
if $e_{x_{spl}} = 254$ **then return** X;

Figure 11.1: Basic pseudo-code on the left and an optimized way on the right for handling special values.

Finally X will be computed as X OR $[00\ldots0]$. Hence, if $x = -0$, it is not changed into a NaN. The idea is to perform either X OR $[00\ldots0]$ or X OR $[11\ldots1]$. It can easily be checked that for all positive x the former is performed, and that for all $x < 0$ the latter is performed. Therefore, x changes into a NaN. After that it is checked whether e_x is 255 or 0 only by performing **if** $e_{x_{spl}} = 254$. If the condition is true X (original or modified) is returned.

11.2.2 Digit Recurrence Algorithms

The underlying principles of the digit recurrence algorithms have already been recalled for division in Section 9.3. Here, the selection function is more complicated than for division as it depends on the partial result too, produced after each iteration. We will discuss here the common nonrestoring algorithm which retires one bit of the result in each iteration.

As said earlier, the mantissa $m_x \in [1, 2)$ might be scaled to the interval $[1, 4)$ to have an even exponent. So, we consider $m_x \in [1, 4)$ and obtain the result as $T \in [1, 2)$. Let $T[j]$ be the result after j iterations, such that

$$T[j] = \sum_{i=0}^{j} t_i r^{-i},$$

where $t_0 = 1$. The final result after n iterations can be written as $T = T[n] = \sum_{i=0}^{n} s_i r^{-i}$. The following recurrence will be used in each iteration:

$$w[0] = m_x - 1, \tag{11.1a}$$

$$w[j] = rw[j-1] - 2T[j-1]t_j - t_j^2 r^{-j}, \quad j = 1, \ldots, n, \tag{11.1b}$$

where $w[j]$ is residual after j iterations. In each iteration, one digit of the result is determined by the selection function

$$t_{j+1} = SEL(w[j], T[j]) \tag{11.2}$$

in such a way so as to bound the residual as

$$|w[j]| < 2T[j] + r^{-j}. \tag{11.3}$$

If all the intermediate residuals are computed correctly then

$$|\sqrt{m_x} - T| < r^{-n},$$

that is, the final result must be precise up to n bits. Moreover, if $w[n] < 0$ then $T = T - r^{-n}$.

As in division, square root uses the quotient digit set $\{-1, +1\}$ and does not restore the correct value in each iteration; instead, it accumulates small errors in one iteration and corrects them in the next iteration. Figure 11.2 shows the nonrestoring algorithm to compute the square root.

Note that the partial residual $w[j]$ depends on the partial result $T[j-1]$ computed in the previous iteration. If $w[n] < 0$, it is made positive and the result is modified. Since $t_0 = 1$, the first iteration is straightforward and can be separated (unrolled) from the loop. So, 24 iterations are required (bit 24 being the guard bit) for correct rounding in case of round to nearest. For round to infinity, a sticky bit is also required.

```
1  t_0 ← 1
2  w[1] ← 2(m_x − t_0) − 2.5
3  T[1] ← 1.5
4  for j from 2 to n do
5      w[j] ← 2 × w[j − 1]
6      if w[j] ≥ 0 then
7          w[j] ← w[j] − (2 × T[j − 1] + 2^{−j})
8          t_j ← 1
9      else
10         w[j] ← w[j] + (2 × T[j − 1] − 2^{−j})
11         t_j ← −1
12     T[j] ← T[j − 1] + t_j 2^{−j}
```

Figure 11.2: Nonrestoring algorithm for square root.

11.2.3 Functional Iteration Algorithms

These algorithms, which have been presented in the division chapter, are more interesting when a full-width multiplier is available since they double the number of quotient bits at each iteration. We have implemented the square root based on the Goldschmidt algorithm. Also note that exactly the same implementation has been used for both processors. We could not try other options to do some further optimizations in the case of ST220 due to time constraints and also because the latency increase is not too much for ST220.

Here, for square root, one iteration of the Goldschmidt algorithm refines an initial approximation to arrive at the root of the mantissa m_x. Square root can be viewed as a *product* of the mantissa m_x and the reciprocal of the square root of m_x, that is,

$$T = \sqrt{m_x} = m_x \times 1/\sqrt{m_x}.$$

An initial approximation $\mathcal{G}_1 = 1/\sqrt{m_x}$ is obtained such that

$$Z_1 = m_x \times \mathcal{G}_1^2 \approx 1$$

and

$$t_1 = m_x \times \mathcal{G}_1 \approx t.$$

The Goldschmidt algorithm [44, 41, 30, 74, 64] consists in finding a sequence $\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3, \ldots$ such that the product $Z_i = m_x \mathcal{G}_1^2 \mathcal{G}_2^2 \ldots \mathcal{G}_i^2$ approaches 1 as i goes to infinity. Hence,

$$t_i = m_x \mathcal{G}_1 \mathcal{G}_2 \ldots \mathcal{G}_i \approx t \quad (i \to \infty).$$

In the implementation of this algorithm, an approximation of $1/\sqrt{m_x}$, where $m_x \in [1, 2)$, is obtained through a degree-3 minimax polynomial. If the unbiased exponent is odd then m_x falls

into a different binade, that is, $m_x \in [2, 4)$. In this case t must be multiplied by $\sqrt{2}$ before final rounding.

As done in division, we split the interval so as to obtain a more accurate polynomial of lower degree. One polynomial is proposed for each interval. The degree-3 polynomial is evaluated as

$$u(m_x) = (u_2 - u_3 m_x)\ m_x^2 + (u_0 - u_1 m_x) \tag{11.4}$$

and its coefficients for each interval are given in Table 11.2 below.

$m_x \in$	coeffs	Decimal	Hexadecimal	Binary
$[1, 3/2)$	u_0	1.9801905...	0x7EBB70F3	01 . 111110...
	u_1	1.6143057	0xCEA191B7	1 . 1001110...
	u_2	0.7842230...	0x64616C1D	0 . 11001000...
	u_3	0.1501597	0x2670DEC4	. 001001100...
$[3/2, 2)$	u_0	1.6636465...	0x6A792F8B	01 . 101010...
	u_1	0.9597583...	0x7AD95CE2	0 . 1111010...
	u_2	0.3310671...	0x2A606850	0 . 01010100...
	u_3	0.0451628	0x0B8FCA7D	. 000010111...

Table 11.2: Coefficients of the polynomial in (11.4). Hex representation is used in the implementation.

The above polynomial gives $\mathcal{G}_1 = 1/\sqrt{m_x}$ such that

$$1 - 2^{-14} \leq Z_1 = m_x \mathcal{G}_1^2 \leq 1 + 2^{-14}.$$

The initial precision of 14 bits (first quotient digit) suffices to perform one iteration. The algorithm can be given as

1 $\mathcal{G}_1 \leftarrow u(m_x)$
2 $Z_1 \leftarrow m_x \times \mathcal{G}_1^2$
3 $t_1 \leftarrow m_x \times \mathcal{G}_1$
4 $\mathcal{G}_2 \leftarrow (3 - Z_1)/2$
5 $t \leftarrow t_1 \times \mathcal{G}_2$
6 $t \leftarrow t \times \sqrt{2}$ **if** $m_x \in [2, 4)$
7 // Correct rounding (compute remainder)
8 **if** $t^2 \geq m_x$ **then**
9 $t \leftarrow t - 1$
10 // Round **to** nearest
11 $T \leftarrow ROUND(t)$.

The multiplications performed here are of type 32×32 so Goldschmidt algorithm is well suited to implement for ST231. Since the same implementation has been used for the ST220 too, these multiplications have been emulated using four $16 \times 32 \rightarrow 32$ multiplications.

11.3 Performances

The performances of floating-point square root, based on two different algorithms, measured both on ST220 and ST231 processors, are given in Table 11.3. Note that, although the implementation of

square root in the original library is also based on the nonrestoring algorithm, our implementation is a bit faster due to optimized handling of special values and unrolling of the first iteration of the loop of the nonrestoring algorithm. Remark that, in the case of ST220, since the same implementation has been used as for ST231, it is expensive due to computation of full-width multiplications.

Different algorithms	Timings (# cycles)	
	ST220	ST231
Original	127	127
Nonrestoring	122	122
Goldschmidt	76	50

Table 11.3: Performances of the implementations based on different algorithms on ST220 and ST231.

Chapter 12

Inverse Square Root

Processors such as Itanium or PowerPC have an instruction which provides a seed for the inverse of square root in order to perform square root in software. The IEEE standard does not provide any support for this operation. Normally, it can be computed by obtaining the square root and then taking its reciprocal (division), which is a approach adopted in the original library. Since it is computed by performing two floating-point operations, the result is obtained after two rounding errors. This kind of implementation may not produce a correctly rounded result. In FLIP, the implementation of this operation is almost the same as the implementation of the square root. The main difference, beacuse of which it is slower than square root, is the emulation of the multiplication of the type $64 \times 32 \rightarrow 32$ in order to have a correctly rounded result. Regarding the performances on ST231, a four fold increase in the latency has been obtained in comparison to the original library. Remark that, since the original library does not support this operation, it must compute it through two floating-point operations (square root followed by reciprocal).

12.1 Floating-Point Inverse Square Root

We must say now that the core of the *fast implementation of square root* in Section 11.2.3 has been inspired from the implementation of the inverse square root since the initial approximation obtained is $1/\sqrt{m_x}$ and not $\sqrt{m_x}$. Indeed we save one multiplication but the 64-bit multiplication and addition completely offset this latency gain, particularly in the case of ST220. After giving a short description of how this operation is computed, since its implementation is the same as of the square root, we will present a few difficulties which hinder further optimizations.

The input for floating-point inverse square root is a single precision floating-point operand x represented by (s_x, e_x, f_x). The implementation of this operation supports no subnormals and only round-to-nearest-even mode. The output is a single precision floating-point number $z = \circ(1/\sqrt{x})$. The mantissa $m_x = 1.f_x$ is formed after the reinsertion of the implicit bit. A short description of this operation is as follows.

1. If x is $\pm\infty$, ± 0 or NaN, return the result as defined in Table 12.1.
2. Inverse square root of the mantissa $IR = 1/\sqrt{m_x}$ is computed.
3. The intermediate result exponent is computed as

$$e_z = 254 - \lfloor (e_x + B)/2 \rfloor.$$

4. The sign of the result is $s_z = 0$.

5. If $m_x \neq 1$, normalize and update the exponent.

6. Round the exact inverse square root IR.

7. Neither underflow nor overflow can occur for the same reasons as were given for square root.

12.1.1 Special Values

In the case where x is a special value the result can be chosen from Table 12.1 below.

x	NaN	$+\infty$	$-\infty$	$+0$	-0
z	NaN	$+0$	NaN	$+\infty$	$-\infty$

Table 12.1: All the possible predefined values of the inverse square root $\circ(1/\sqrt{x})$ when x is a special value.

Handling of special values is done in the following way.

1. Zero operands are filtered in the trivial computation step. The result returned is $\pm\infty$ depending on the sign of the zero.

2. All $x < 0$ are filtered as described in Section 11.2.1.

3. After this, the pseudo-code below helps to filter other special values.

```
if e_x = 255 then
  if f_x ≠ 0 then return NaN;
return ±0;
```

12.2 Inverse Square Root of Mantissa

After handling special values the inverse square root of the mantissa m_x is computed. The intermediate result exponent and the sign of the result can easily be obtained. As said earlier in the case of the implementation based on the Goldschmidt algorithm, the *square root* of the mantissa is obtained by first computing the approximation of the inverse square root. The Goldschmidt algorithm [44, 30] with the same polynomial (11.4) for initial approximation has been used. So, for basic details see Section 11.2.3. From the initial approximation $\mathcal{G}_1 = 1/\sqrt{m_x}$ we have 14 bits of precision, that is,

$$1 - 2^{-14} \leq Z_1 = m_x \mathcal{G}_1^2 \leq 1 + 2^{-14}.$$

It therefore suffices to perform only one iteration of Goldschmidt. The algorithm is

```
1  G1 ← u(mx)
2  Z1 ← mx × G1^2
3  ir1 ← G1
4  G2 ← (3 − Z1)/2
5  ir ← ir1 × G2
6  ir ← ir × (1/√2)      if mx ∈ [2, 4)
7  // Correct rounding (compute the remainder)
8  if ir^2 × mx ≥ 1 then
9      ir ← ir − 1
10 // Rounding (nearest even)
11 IR ← ROUND(ir).
```

12.3 Rounding

The most difficult part in the implementation of inverse square root is to obtain the correctly rounded result. The function ROUND is called for correct rounding. This function only verifies whether $ir^2 \times m_x \geq 1$. This verification consists of two multiplications, $ir \times ir$ and $ir^2 \times m_x$. Since ir has 28 bits and an exact ir^2 is required, ir^2 must be stored as a 64-bit value (two 32-bit registers). Now, performing the 64×32 multiplication $ir^2 \times m_x$ is very expensive on the target processors. Since the available multiplier on ST231 is $32 \times 32 \rightarrow 32$, this rather expensive multiplication must be splitted and performed as two consecutive 32×32 multiplications. The intrinsic `__mulun` has been used which computes a 64-bit product of two unsigned 32-bit values. Remark that, for ST220, since the same implementation has been used as for ST231, it is more expensive due to the availability of a rectangular multiplier $16 \times 32 \rightarrow 32$. In all, rounding requires three sequential multiplications (four if $ir \times 1/\sqrt{2}$ is performed) or around 9 (12) cycles.

12.4 Performances

The performances of floating-point inverse square root have been measured on ST220 and ST231 processors and are given in Table 12.2. Note that the original library supports this operation by performing two floating-point operations, square root followed by division. That is why the timing is a sum of the timings for both operations individually. The speed-up of our implementation is due to optimizations done at the algorithmic level.

Implementation	Timings (# cycles)	
	ST220	ST231
Original	306	304
Goldschmidt	96	67

Table 12.2: Performances of the implementations based on different algorithms on ST220 and ST231.

Part IV

Some Elementary Functions

Chapter 13

Introduction to Elementary Functions

Elementary functions are used in many application areas of scientific computing and computer graphics. Fast evaluation of these functions is vital to the performance of many DSP applications, particularly when a short word length (16–24 bits) is involved. Elementary functions are categorized into *trigonometric* (sin, cos, tan), *inverse trigonometric* ($\sin^{-1}$), *exponential* (e^x, 2^x), *logarithm* (ln, $\log_2$) and other *transcendental* functions. In FLIP, we are interested in $\sin(x)$, $\cos(x)$, $\log_2(x)$ and 2^x.

13.1 Implementation of Elementary Functions

Concerning the accuracy of the results of the basic arithmetic operations ($\pm, \times, /, \sqrt{}$), the IEEE standard specifies the property of correct rounding as we have seen in Section 1.5. Unfortunately, the standard does not specify anything about the correct rounding of the elementary functions. This is due to the fact that correctly rounded results can not be easily obtained as dictated by the *Table Maker's Dilemma* [43, 52]. In general, these functions cannot be exactly computed by performing a finite number of basic arithmetic operations. It is therefore natural to approximate them. Hence, the accuracy of the results of the elementary functions is governed by the error of the approximation and by the rounding errors that occur during the evaluation of the approximation. However, for limited precision, such as single precision representation, it is possible to have practical implementations that produce correctly rounded results. Software routines provided in [13] always produce correctly rounded results for single precision exponential.

From [68] we know that correct rounding of elementary functions does not only minimize the distance between the computed result and the exact value of the function but also preserves important mathematical properties such as *monotonicity* and *symmetry* [86, 13, 42]. The computation of these functions can be performed in software through software libraries as well as in hardware through a partial software support. In software, computation is based on the techniques such as polynomial or rational approximations [20, 35]. As cited in [87], which also provides an extensive list of application domains of elementary functions, implementation of elementary functions in processors such as the AMD K5 [62], the Cyrix 83D87 [39], and the Intel Pentium [12] is based on table-driven methods. Other techniques generally employed in hardware are parallel polynomial approximations [80, 37], rational approximations, and shift-and-add algorithms, such as CORDIC [94, 49, 54]. The CORDIC algorithm was introduced by Volder [94] for computing trigonometric functions, and generalized by Walther [95]. It has been implemented in many machines, for example, in Hewlett Packard's HP 35 and Intel 8087. The choice of the technique and its implementation depends on many parameters such as the number of bits of the argument and the result required as well as the requirement to achieve the correct last bit.

13.1.1 Table Maker's Dilemma

The presented information here has been inspired from [68]. If a function f (sine, cosine, exponential, logarithm...) is to be evaluated then an approximation to f must be computed. Let x be a floating-point number in any given format with a b-bit mantissa. We want to compute $\diamond(f(x))$ by implementing the function f with correct rounding and produce the result in the same format. We compute $F(x)$, an approximation to $f(x)$, with extra accuracy and round $F(x)$ according to the active rounding mode. It is not possible that rounding $F(x)$ will produce the final result which is equivalent to the correct rounding of $f(x)$ if $f(x)$ has the form:

- in rounding towards $+\infty$, $-\infty$ and 0 modes,

$$\overbrace{\underbrace{1.XXX\cdots XXX}_{b \text{ bits}}000000\cdots000}^{m \text{ bits}}XXX\cdots$$

  or

$$\overbrace{\underbrace{1.XXX\cdots XXX}_{b \text{ bits}}111111\cdots111}^{m \text{ bits}}XXX\cdots;$$

- in rounding to the nearest mode,

$$\overbrace{\underbrace{1.XXX\cdots XXX}_{b \text{ bits}}1000000\cdots000}^{m \text{ bits}}XXX\cdots$$

  or

$$\overbrace{\underbrace{1.XXX\cdots XXX}_{b \text{ bits}}0111111\cdots111}^{m \text{ bits}}XXX\cdots.$$

This impossibility of producing a correctly rounded result is known as Table Maker's Dilemma. It forces us to find out the lowest accuracy (smallest m) with which to compute $F(x)$ in order to produce the correctly rounded value of a function, in a given domain and for a particular rounding mode. This can be done by finding the "worst cases" for any function, that is, finding a floating-point number x such that $f(x)$ is closest to another floating-point number (in case of rounding towards $+\infty$, $-\infty$ and 0 modes) or the exact middle of two consecutive floating-point numbers (in case of rounding to the nearest mode). Lefèvre, in his PhD thesis [61], proposed an algorithm to find worst cases for some functions. This algorithm helped to find the smallest value of m required to compute $F(x)$ for the functions e^x, $\log(x)$ and 2^x where x is a double precision floating-point number. Schulte and Swartzlander [79, 80] proposed algorithms to compute $1/x$, $\sqrt{x}$, 2^x and $\log_2(x)$ correctly rounded in single precision. They performed an exhaustive search for $b = 16$ and $b = 24$ to find the smallest m. For $b = 16$, they found $m = 35$ for $\log_2(x)$ and $m = 29$ for 2^x, and for $b = 24$, they found $m = 51$ for $\log_2(x)$ and $m = 48$ for 2^x. It can easily be assumed that $m \approx 2b$. So, for such precisions correct rounding is possible but is quite expensive. For higher precisions such as double and quad, the reader is invited to consult Muller's book [68] which provides an explanation for always getting the value of m around $2b$ (or slightly above $2b$) in practice. It also provides the results of the Lefèvre's experiments of finding the worst cases of the Table Maker's Dilemma. Finding out these worst cases has really helped in providing correct rounding of elementary functions. The libraries Libmcr [8] and CRlibm [1] provide correct rounding and have good performances too.

13.1.2 Hardware Implementation

Assuming that x takes only a small set of values, the simplest hardware implementation to compute the function $f(x)$ consists in storing the values of $f(x)$ in a table for all possible arguments. However, the size of the table increases exponentially in terms of the width of the input values. Hence, this approach is limited by the available table size.

For general purpose processors, table lookup algorithms have already been used to compute the initial approximations to reciprocal and inverse square root in order to implement division and square root through Newton-Raphson or Goldschmidt iterations.

The state-of-the-art methods are based on multipartite tables which were introduced in 1995 [78] and were improved later [66, 87, 25]. Other methods consist of the table based methods with an addition [46]. To know more about hardware implementation see [68] and the references given therein. We will now focus on the *software implementation* of elementary functions which is the case concerning FLIP and this document.

13.1.3 Software Implementation

The table based methods are useful for small precision input values. For larger precision, the resulting table is impractical to implement and so we look for algorithms which are based on polynomial approximations. The quality of the approximation is judged by speed, accuracy and cost. How fast an approximation can be computed determines the speed. The error incurred during the evaluation of a polynomial decides the accuracy. These approximation methods only apply to a small domain of the argument. So, an initial step, called range reduction (see Section 13.2), is included before the evaluation of a polynomial is done in this reduced domain and finally the result is obtained after the reconstruction step. Generally, these approximations utilize only basic operations, such as addition, multiplication and, in order to reduce the degree of the polynomial, table lookup. Since tables cannot be used on our target architecture (see Section 9.3.1 for reasons), we will focus on software implementation of elementary functions based on polynomial approximations which uses only additions and multiplications. Also, note that the implementation of $\sin(x)$, $\cos(x)$, $\log_2(x)$ and 2^x in FLIP only supports normalized numbers for the round to nearest mode. Moreover, the implementation has been targeted only for the processor ST231. For any floating-point x the computation of the function $f(x)$ can be performed in three steps as follows.
Reduction: In order to compute $f(x)$, x is mapped to x^* such that x^* is bounded by the convergence domain $[L, U]$. This mapping is called *reduction transformation*.
Approximation: The function f is now computed for the transformed input, that is, $f(x^*)$ is computed through an approximation formula

$$p(x^*) \approx f(x^*).$$

Generally and in our case, p is a polynomial.
Reconstruction: Finally, a second transformation is done which compensates the original transformation in the reduction step to compute $f(x)$ which is valid for the complete domain of the argument.

We will now detail these steps as well as several methods for performing these steps in Sections 13.2 and 13.3.

13.2 Range Reduction

It might be possible to approximate a function for the complete interval using a polynomial of very high degree. But, evaluation of such a polynomial will excessively increase the computation.

So, using a single polynomial for the complete interval is not an efficient solution. Therefore the approximation step is performed in a small domain of the argument. Table 13.1 gives N, the number of significant bits (obtained as $-\log_2(\text{absolute error})$) of the minimax approximations (see Section 13.3.1) to the function 2^x on different domains of the argument by polynomial of degree 3. The fall in the accuracy (1.06 bits) for larger domains such as $[-2^2, 2^2]$ makes it impossible to use approximation methods for the complete domain of the argument. Therefore the domain must be reduced. For software implementation of elementary functions, the range reduction step is very important since the latency and accuracy of this step influence the overall performance of the algorithm being implemented and the accuracy of the result respectively. The reduction transformation allows to deduce $f(x)$ from $f(x^*)$, where

- x^* is called a *reduced argument* and is deduced from x;
- x^* belongs to the convergence domain $[L, U]$ of the algorithm implemented for the evaluation of f.

m	N
-3	21.69
-2	17.69
-1	13.68
-0	9.65
1	5.53
2	1.06

Table 13.1: Number of significant bits (N) of the minimax approximations to the function 2^x on the domain $[-2^m, 2^m]$ by a degree-3 polynomial.

Range reduction methods, as described in [68], can be classified into two classes as follows:

1. *Additive reduction*, generally employed for trigonometric functions, in which x^* is written as $x - kC$, where k is an integer and C is an integer multiple of $\pi/4$ in case of trigonometric functions. In our implementation of the functions sine and cosine, the convergence domain used to compute the sine and cosine of the reduced argument is $[0, \pi/4]$. We choose $C = \pi/2$ and compute x^* and k as

$$k = \left\lfloor x \times \frac{1}{C} \right\rfloor,$$

and

$$x^* = x - k\pi/2,$$

which gives

$$x^* \in [0, \pi/4].$$

After the reduction step, if k is even, computing $\sin(x)$ and $\cos(x)$ will be equivalent to computing $\sin(x^*)$ and $\cos(x^*)$ and will be equivalent to $\cos(x^*)$ and $\sin(x^*)$ if k is odd. In case of negative values of x, we follow the identities $\cos(-x) = \cos(x)$ and $\sin(-x) = -\sin(x)$. So for these functions such as sine and cosine no reconstruction step is required.

2. *Multiplicative reduction*, generally employed for logarithm and exponential, in which x^* is written as x/C^k, where k is an integer and C is a constant. In our implementation of the function logarithm, the convergence domain used to compute $\log_2(x^*)$ is $[1, 2]$. We choose $C = 2$ and k as e_x, the exponent of the floating-point number x. We then compute

$$x^* = x/2^k,$$

which gives

$$x^* \in [1, 2].$$

After the reduction step, $\log_2(x^*)$ is computed and the reconstruction step gives the final result as

$$\log_2(x) = \log_2(x^*) + k.$$

For the moment, the sine and cosine functions are only implemented for the domain $[0, \pi/4]$ in FLIP.

13.3 Polynomial Approximation

In the implementation of elementary functions, polynomial approximation is the second step after the range reduction step. We assume that the domain has been reduced so now we focus on approximating a function f in the domain $[L, U]$. In this domain we try to approximate the function f through a polynomial. There exist different types of polynomials. A particular polynomial is chosen depending on the domain of the function, demand to reduce the error and the implementation issues. As described in [68], polynomials can be classified into two kinds: the ones that minimize the maximum absolute error over a given interval, called *minimax approximation*, and ones that minimize the average error over a given interval, called *least squares approximation*. The total error incurred is determined by the approximation error and the evaluation error (roundoff errors propagate since only finite precision is available). Since we want to minimize the absolute error, we consider minimax polynomial approximation in our implementations.

13.3.1 Minimax Approximation

Let $\mathcal{P}_n$ be the set of real polynomials of degree at most n. Now, we want to approximate a function f by a degree-n polynomial $p^* \in \mathcal{P}_n$ on the interval $[a, b]$. The distance $||f - p||_\infty$ can be given as:

$$||f - p||_\infty = \max_{a \le x \le b} |f(x) - p(x)|.$$

A minimax polynomial approximation p^* satisfies:

$$||f - p^*||_\infty = \min_{p \in \mathcal{P}_n} ||f - p||_\infty.$$

Another theorem by Chebyshev shows that if $f(x)$ is continuous on the interval $[a, b]$ then there exists a unique minimax polynomial for a particular n and the maximum error is attained on at least $n + 2$ distinct points in the interval $[a, b]$ and the sign of the error alternates. The reader is invited to consult [68] for more details. This minimax approximation is based on the Remez algorithm [75].

In our implementation, we have obtained minimax approximations for various functions through Maple [55]. Before using the minimax command one must include the "numapprox" package as

```
> with(numapprox);
```

The numapprox package contains commands for the numerical approximation of functions. The command

```
> minimax(f(x), x=a..b, [p,q], w(x), 'err');
```

returns the best rational approximation, with a degree-p numerator and a degree-q denominator, of the function $f(x)$ on the interval $[a, b]$ with a weight function $w(x)$ and the approximation error stored in `err`. In our case we are interested only in polynomials of the form

$$p(x) = p_0 + p_1 x + p_2 x^2 + \cdots + p_n x^n.$$

So, we set $q = 0$ in the above command. The polynomial returned by Maple is an approximation of the exact minimax polynomial. The evaluation of minimax polynomials is done in the same way as discussed in Section 9.4.1.1.

13.4 FDlibm

In this section we detail the key features of the library used by STMicroelectronics to support elementary functions on the target processors. FDlibm (Freely Distributable libm) [4] is a C library of mathematical functions developed for machines that support the IEEE-754 standard for floating-point arithmetic. Apart from the four elementary functions which have been supported in FLIP, it also provides support for various inverse trigonometric and hyperbolic functions, gamma function, power (x^y), cube root, etc. All the operations are supported for double as well as single precision floating-point numbers. Most of the routines provided in this library are coded directly from the algorithms and the coefficients given in [20]. So this library has not been optimized for any particular processor. Unlike FLIP 0.3, this library performs range reduction step for the functions sine and cosine.

In order to be able to use FDlibm on the target processors, it must be used in conjunction with another library which provides support for the basic floating-point operations ($\pm, \times, /$). So STMicroelectronics used FDlibm with their original library which was optimized for the target processors. Whenever there is an operation, let us say multiplication, performed in FDlibm either in the range reduction step or polynomial approximation, it is transformed into a call for the function `__muls` in the original library. Concerning accuracy, FDlibm provides an error less than 1 ulp for important functions such as sine, cosine, exponential and logarithm. Moreover, for the power operation (x^y), it provides almost always a correctly rounded result. That is why the base-2 exponential is computed through the power operation. See Sections 14.3 and 15.3 for a performance comparison between FDlibm and FLIP 0.3.

Chapter 14

Sine and Cosine

14.1 Floating-Point Sine and Cosine

The input for sine and cosine is a single precision operand x represented by (s_x, e_x, f_x). The output is a floating-point number $z \approx f(x)$, where f is either the sine or the cosine function. The basic description of the algorithm can be given as follows:

1. If x is $\pm\infty$, ± 0 or NaN, return the result as defined in Table 14.2.
2. Perform the range reduction step to obtain the reduced argument x^*. In our case $x^* \in [0, \pi/4]$.
3. Evaluate a polynomial approximating $\sin(x^*)$ or $\cos(x^*)$ in this interval.
4. Reconstruct $\sin(x)$ or $\cos(x)$ as discussed in Section 13.2.

14.2 Algorithm and Implementation

Since the range reduction step for sine and cosine has not been implemented, we present straightway the respective polynomials approximating these functions for the reduced interval $[0, \pi/4]$ and their implementations tuned to the target architecture. A major problem in approximating $\sin(x^*)$ is that when x^* is very close to 0, one looses all the significant bits in the result. In order to avoid such a situation one considers an approximation to $\sin(x^*)/x^*$ and multiply it later by x^*. This allows to preserve the significant bits and thus, guards the accuracy. We have tried minimax polynomials of different degrees to approximate $\sin(x^*)/x^*$. The number of significant bits obtained for the interval $[0, \pi/4]$ for each polynomial is given in Table 14.1. The reason behind trying only the odd degree polynomials is the same as the one discussed in Section 9.4.1.1. Note that this is only a theoretical estimation of the number of significant bits. The exact number of bits can be obtained once the approximation and the evaluation errors are measured. We chose the degree-5 polynomial mainly due to following reasons.

1. Our aim was not to produce a correctly rounded result. Since the applications involving elementary functions do not require the last-bit accuracy, as decided with STMicroelectronics our goal was to produce a sufficiently accurate result (a difference of at most 3 ulps from the exact result) without degrading the speed. Of course, if the last-bit accuracy is at stake, speed must be compromised.
2. During validation we have compared, for each single precision floating-point number in the domain $[0, \pi/4]$, the result produced by FLIP with the one produced by the libm of GCC.

We conclude that our result is at a distance of at most 1 ulp from the result of this libm. Assuming that this libm produces a result which is at most 1 ulp from the exact result, we can say that our goal has been fulfilled.

Function / Degree	1	3	5	7
$\sin(x^*)/x^*$	6.35	15.38	25.48	36.35
$\cos(x^*)$	4.82	13.10	22.70	33.20

Table 14.1: Number of theoretical significant bits of the minimax approximations to the sine and cosine functions on the interval $[0, \pi/4]$ by polynomials of different degrees.

The philosophy of implementing the cosine function is the same as the sine function with only the two following differences.

1. Unlike evaluating $\sin(x^*)/x^*$, $\cos(x^*)$ is evaluated over the complete interval $[0, \pi/4]$.
2. As evident from the last row of Table 14.1, the degree-5 polynomial used so far does not provide enough accuracy to fulfill our goal. So instead of going for a higher degree polynomial, we split the complete interval into two small intervals, as discussed in Section 9.4.1.2 for division, to gain a few bits of accuracy without having to go beyond degree 5. After splitting, we have 28.60 significant bits in the interval $[0, \pi/8]$ and 28.64 bits in $[\pi/8, \pi/4]$.

We now present the handling of special values followed by the polynomial approximation for sine and cosine functions in Sections 14.2.2 and 14.2.3.

14.2.1 Special Values

Refer to Table 14.2 below.

x	NaN	$\pm\infty$	± 0
$z = \sin(x)$	NaN	NaN	0
$z = \cos(x)$	NaN	NaN	1

Table 14.2: All the possible predefined values of the sine and cosine when x is a special value.

14.2.2 Polynomial Approximation for Sine

The degree-5 polynomial used to approximate $\sin(x^*)/x^*$ on the interval $[0, \pi/4]$ is

$$p(x^*) = (-p_5 x^* + p_4)\ {x^*}^4 + (-p_3 x^* - p_2)\ {x^*}^2 - p_1 x^* + p_0, \tag{14.1}$$

where the coefficients are given in Table 14.3

The approximation $p(x^*)$ is multiplied by x^* to produce $\sin(x^*)$. The reconstruction step is performed as explained in Section 13.2. Rounding is performed and the result exponent is decremented in order to normalize the result (since $\sin(x^*)$ is always less than 1).

$x^* \in$	Coeffs	Decimal	Hexadecimal	Binary
$[0, \pi/4)$	p_0	$1.000000021350\ldots$	0x8000002E	$1\ .\ 00\ldots$
	p_1	$0.196667141550\ldots \times 10^{-5}$	0x83FB285C	$.\ 1000\ldots \times 2^{-18}$
	p_2	$0.166637303803\ldots$	0xAAA2F829	$.\ 1010\ldots \times 2^{-2}$
	p_3	$0.160495915856\ldots \times 10^{-3}$	0xA84ACB5B	$.\ 1010\ldots \times 2^{-12}$
	p_4	$0.873075894862\ldots \times 10^{-2}$	0x8F0B750A	$.\ 1000\ldots \times 2^{-6}$
	p_5	$0.455778465675\ldots \times 10^{-3}$	0xEEF58CD6	$.\ 1111\ldots \times 2^{-11}$

Table 14.3: Coefficients of the minimax polynomial in (14.1) approximating $\sin(x^*)/x^*$.

14.2.3 Polynomial Approximation for Cosine

We compute an approximation to $\cos(x^*)$ by using two degree-5 polynomials for two different intervals $[0, \pi/8)$ and $[\pi/8, \pi/4]$. The degree-5 polynomial is

$$q(x^*) = (-q_5 x^* + q_4)\ x^{*4} + (-q_3 x^* - q_2)\ x^{*2} - q_1 x^* + q_0 \tag{14.2}$$

where the coefficients for the two intervals are given in Table 14.4

$x^* \in$	Coeffs	Decimal	Hexadecimal	Binary
$[0, \pi/8)$	q_0	$1.00000000244\ldots$	0x80000005	$1\ .\ 00\ldots$
	q_1	$0.448251196951\ldots \times 10^{-6}$	0xF0A72CE7	$.\ 1111\ldots \times 2^{-21}$
	q_2	$0.499986665314\ldots$	0xFFFE4090	$.\ 1111\ldots \times 2^{-1}$
	q_3	$0.145121298568\ldots \times 10^{-3}$	0x982BB3B3	$.\ 1001\ldots \times 2^{-12}$
	q_4	$0.423810618082\ldots \times 10^{-1}$	0xAD97C3A7	$.\ 1010\ldots \times 2^{-4}$
	q_5	$0.162341273808\ldots \times 10^{-2}$	0xD4C8B13C	$.\ 1101\ldots \times 2^{-9}$
$[\pi/8, \pi/4]$	q_0	$1.00004247646\ldots$	0x80016452	$1\ .\ 000\ldots$
	q_1	$0.461928594497\ldots \times 10^{-3}$	0xF22F01A7	$.\ 1111\ldots \times 2^{-11}$
	q_2	$0.497927504717\ldots$	0xFEF05A99	$.\ 1111\ldots \times 2^{-1}$
	q_3	$0.491751880871\ldots \times 10^{-2}$	0xA123233B	$.\ 1010\ldots \times 2^{-7}$
	q_4	$0.481933294976\ldots \times 10^{-1}$	0xC5665E61	$.\ 1100\ldots \times 2^{-4}$
	q_5	$0.462289201346\ldots \times 10^{-2}$	0x977BA101	$.\ 1001\ldots \times 2^{-7}$

Table 14.4: Coefficients of the minimax polynomial in (14.2) approximating $\cos(x^*)$.

Note that, if x^* is very close to zero, then post-normalization is required and the exponent is updated.

14.3 Performances

We have measured the performances, in number of cycles, of the sine and the cosine functions in three different ways as:

1. FDlibm in conjunction with the original library from STMicroelectronics;

2. FDlibm in conjunction with FLIP for basic and additional operations, that is, FLIP 0.2;

3. FLIP with support for elementary functions, that is, FLIP 0.3.

The performances are given in Table 14.5. Evaluation of both functions in FDlibm involves 25–30 function calls to floating-point operations for each function. Moreover FLIP 0.3 evaluates these functions for the reduced argument x^*. So, it it not surprising that FLIP 0.3 has a speed-up factor of about 40.

Library	$\sin(x)$	$\cos(x)$
FDlibm + Original	1249	1403
FDlibm + FLIP 0.2	842	958
	$\sin(x^*)$	$\cos(x^*)$
FLIP 0.3	29	36

Table 14.5: Performances of the sine and the cosine functions in three different ways as measured on the processor ST231.

When the first way is compared with the second way of measuring the performances, we find speed-up factors of 1.48 and 1.46 for the sine and cosine functions respectively. The improvements are due to the fact that FLIP 0.2 is faster than the original library for each basic and/or additional operation (see Table 3.4). We find that the speed-up factors for both functions lie between the speed-up factors of addition/subtraction, multiplication and FMA/FMS. It is because FDlibm evaluates a polynomial in the Horner fashion and so there are approximately 6 calls to the FMA/FMS operation in the polynomial approximation step.

Chapter 15

Base-2 Exponential and Logarithm

15.1 Floating-Point Exponential and Logarithm

The input for exponential and logarithm is a single precision operand x represented by (s_x, e_x, f_x). The output is a floating-point number $z \approx f(x)$, where f is either the exponential or the logarithm function, in base-2. The basic description of the algorithm can be given as follows:

1. If x is $\pm\infty$, ± 0 or NaN, return the result as defined in Table 15.3.
2. Perform the range reduction step on x to find the reduced argument x^*. In our case the convergence domain is $[1, 2)$ for logarithm and $[0, 1)$ for exponential.
3. Evaluate a polynomial approximation to 2^{x^*} or $\log_2(x^*)$ in there respective intervals.
4. Reconstruct 2^x or $\log_2(x)$.
5. Determine exceptions by verifying the magnitude of the argument.

15.2 Algorithm and Implementation

For exponential the additive range reduction method is used. We choose $C = 1$ and compute x^* and k as

$$k = \lfloor x \rfloor$$

and

$$x^* = x - kC,$$

which gives

$$x^* \in [0, 1).$$

After the reduction step, 2^{x^*} is computed and the reconstruction step gives the final result as

$$2^x = 2^{x^*} \times 2^k.$$

One can see that 2^{x^*} is the normalized mantissa of the result since it lies in the interval $[1, 2)$ and k is the result exponent.

For the implementation of the reduction step we compute the integers k and x^* and store them in two different 32-bit registers. Storing might be a problem, for example if x is of the order 2^{33}. But now, evaluation of 2^x will result in an overflow so we never have to worry about such large numbers. To be precise, we check two conditions as follows. If $x > 1.11\ldots1 \times 2^6$, overflow occurs

in which case the result returned is $+\infty$. Moreover, if $x \leq -126$, underflow occurs and since denormalized numbers are not supported, the result returned is zero.

Let us denote x^* and k by x_I (integral part) and x_F (fractional part) respectively. Now x can be written as $x_I.x_F$ and 2^x can be expressed as

$$2^{x_I.x_F} = 2^{x_I} 2^{x_F}.$$

It entails to find 2^{x_I} to produce the final result. We use a minimax polynomial to approximate 2^{x_F} as presented in Section 15.2.2.

The range reduction step for the logarithm has already been discussed in Section 13.2. A minimax polynomial has been used to approximate $\log_2(x^*)$ as presented in Section 15.2.3. Let us denote x^* by m_x the mantissa which still lies in the convergence domain $[1, 2)$. The mantissa can be expressed as $1 + f_x$ where f_x is the fraction of the floating-point number x. Now, we try to compute $\log_2(1 + f_x)$ where $f_x \in [0, 1)$ instead of $\log_2(m_x)$ with $m_x \in [1, 2)$. The main reason behind doing this is that almost all coefficients lie in the same binade as can be seen in Table 15.5 of the coefficients of a polynomial. Having the coefficients in the same binade allows to avoid some scaling operations during the evaluation of the polynomial. Moreover, the accuracy in both cases is exactly the same.

We have tried minimax polynomials of different degrees to approximate these functions. The number of significant bits obtained for their respective intervals for each polynomial is given in Table 15.1. Again, instead of choosing a polynomial of degree 9, we choose a degree-7 polynomial but split the intervals in order to increase the accuracy. We split the intervals into two small intervals $[0, \sqrt{2} - 1)$ and $[\sqrt{2} - 1, 1)$. This unconventional splitting of intervals can be justified by the statistics given in Table 15.2. This table provides the number of significant bits obtained after splitting the intervals, through conventional and unconventional ways, for each polynomial approximating $\log_2(1 + f_x)$. We find that to have almost the same number of significant bits (29.72), either we must try higher degree polynomial (degree-8 is sufficient) or increase the number of intervals, which will increase the computation, to have the same degree polynomial.

Function / Degree	1	3	5	7	9
2^{x_I}	4.53	13.18	23.15	34.02	45.57
$\log_2(1 + f_x)$	4.53	10.61	16.28	21.78	27.18

Table 15.1: Number of significant bits of the minimax approximations to the exponential and logarithm functions on the interval $[0, 1)$ by polynomials of different degrees.

Like the sine function, computing $\log_2(1 + f_x)$ gives troubles when $f_x \approx 0$. Since $\log_2(m_x) \approx 0$, one looses all the significant bits of the result. In order to avoid such a situation one computes another polynomial to approximate $\log_2(m_x)/(m_x - 1)$ only for the interval $0.96875\ldots \leq m_x \leq 1.03125$. All the values close to 1 such that

$$1 - 2^{-5} < x < 1 + 2^{-5}$$

are treated in this way. Once the polynomial is evaluated, it is multiplied by $m_x - 1$ to produce $\log_2(m_x)$. We evaluate a minimax polynomial of degree 3 for the interval $[lb, ub]$ where lb and ub are the real bounds which are

$$lb = 0.981131076812744140625$$

and

$$ub = 1.021895527839660644531 25.$$

Intervals / Degree	5	7	8	9
$[0, 1)$	16.28	21.78	24.49	27.18
$[0, 1/2)$	20.88	27.91	31.39	34.85
$[1/2, 1)$	23.85	31.86	35.83	39.78
$[0, 1/4)$	26.04	34.79	39.12	43.44
$[1/4, 1/2)$	27.79	37.12	41.74	46.35
$[1/2, 3/4)$	29.24	39.05	43.92	48.77
$[3/4, 1)$	30.48	40.71	45.78	50.84
$[0, \sqrt{2}-1)$	22.24	29.72	33.42	37.10
$[\sqrt{2}-1, 1)$	22.24	29.72	33.42	37.10

Table 15.2: Number of significant bits after splitting the intervals for each polynomial approximating $\log_2(1 + f_x)$.

The number of significant bits obtained from this minimax polynomial is 27.26.

We now present the handling of special values followed by the polynomial approximation for these functions.

15.2.1 Special Values

Refer to Table 15.3 below.

x	NaN	$+\infty$	$-\infty$	$+0$	$x < 0$
$z = 2^x$	NaN	$+\infty$	0	1	$\approx 2^x$
$z = \log_2(x)$	NaN	$+\infty$	NaN	$-\infty$	NaN

Table 15.3: All the possible predefined values of the exponential and the logarithm when x is a special value.

15.2.2 Polynomial Approximation for the Exponential

The degree-7 polynomial which approximates 2^{x_F} on the interval $[0, 1)$ is

$$p(x_F) = ((p_7 x_F + p_6)\ x_F^2)\ x_F^4 + (p_5 x_F + p_4)\ x_F^4 + (p_3 x_F + p_2)\ {x_F}^2 + p_1 x_F + p_0 \tag{15.1}$$

where the coefficients are given in Table 15.4. Note that this polynomial is evaluated in the same order as given in (15.1). This way of evaluation takes 11 cycles.

15.2.3 Polynomial Approximation for the Logarithm

The degree-7 polynomial which approximates $\log_2(1 + f_x)$ for two different intervals is

$$q(f_x) = ((q_7 f_x - p_6)\ {f_x}^2)\ f_x^4 + (q_5 f_x - q_4)\ f_x^4 + (q_3 f_x - q_2)\ {f_x}^2 + q_1 f_x + q_0 \tag{15.2}$$

where the coefficients for two intervals are given in Table 15.5

$x_F \in$	Coeffs	Decimal	Hexadecimal	Binary
$[0, 1)$	p_0	$0.999999999942\ldots$	0x7FFFFFFF	$0\ .\ 111\ldots$
	p_1	$0.693147187818\ldots$	0x58B90C0B	$0\ .\ 1011\ldots$
	p_2	$0.240226356040\ldots$	0x7AFEF2EE	$0\ .\ 111\ldots \times 2^{-2}$
	p_3	$0.055505302351\ldots$	0xE3598726	$.\ 1110\ldots \times 2^{-4}$
	p_4	$0.009613506100\ldots$	0x9D81F793	$.\ 1001\ldots \times 2^{-6}$
	p_5	$0.001343024522\ldots$	0xB0086CCD	$.\ 1011\ldots \times 2^{-9}$
	p_6	$0.000142962416\ldots$	0x95E82E7D	$.\ 1001\ldots \times 2^{-12}$
	p_7	$0.000021660750\ldots$	0xB5B41B75	$.\ 1011\ldots \times 2^{-15}$

Table 15.4: Coefficients of the minimax polynomial in (15.1) approximating 2^{x_F}.

$f_x \in$	Coeffs	Decimal	Hexadecimal	Binary
$[0, \sqrt{2}-1)$	q_0	0	0x0	$0\ .\ 0$
	q_1	$1.442694677139\ldots$	0xB8AA381C	$1\ .\ 011\ldots$
	q_2	$0.721328259049\ldots$	0xB8A8F802	$.\ 1011\ldots$
	q_3	$0.480506981846\ldots$	0x7B02816D	$.\ 0111\ldots$
	q_4	$0.356711313772\ldots$	0x5B516EC3	$.\ 0101\ldots$
	q_5	$0.266130996148\ldots$	0x44212935	$.\ 0100\ldots$
	q_6	$0.166054114235\ldots$	0x2A8285BE	$.\ 0010\ldots$
	q_7	$0.058480575162\ldots$	0x0EF8953E	$.\ 00001\ldots$
$[\sqrt{2}-1, 1)$	q_0	$0.000176543303\ldots$	0x000B91E8	$.\ 0000$
	q_1	$1.440342535942\ldots$	0xB85D24EB	$1\ .\ 0111\ldots$
	q_2	$0.707496256997\ldots$	0xB51E7986	$.\ 1011\ldots$
	q_3	$0.433185694548\ldots$	0x6EE541F7	$.\ 0110\ldots$
	q_4	$0.252889486164\ldots$	0x40BD5D89	$.\ 0101\ldots$
	q_5	$0.117256226777\ldots$	0x1E04810B	$.\ 0001\ldots$
	q_6	$0.035744257689\ldots$	0x09268922	$.\ 00001\ldots$
	q_7	$0.005169001408\ldots$	0x0152C174	$.\ 00000001\ldots$

Table 15.5: Coefficients of the minimax polynomial in (15.2) approximating $\log_2(1 + f_x)$.

In case when $f_x \approx 0$, the degree-3 polynomial which approximates $\log_2(m_x)/(m_x - 1)$ in the interval $[lb, ub]$ is

$$v(m_x) = (-v_3 m_x + v_2)\ {m_x}^2 - v_1 m_x + v_0, \tag{15.3}$$

where the coefficients are given in Table 15.6

The approximation $q(f_x)$ or $v(m_x)$ is rounded to produce the final result.

15.3 Performances

We have measured the performances (in number of cycles) of the exponential and the logarithm functions in three different ways as shown in Table 15.7.

$m_x \in$	Coeffs	Decimal	Hexadecimal	Binary
$[lb, ub]$	v_0	$3.004138513070\ldots$	0xC043CE2F	$0\ .\ 111\ldots$
	v_1	$2.760620495787\ldots$	0xB0AE0197	$0\ .\ 1011\ldots$
	v_2	$1.558258414692\ldots$	0xC7750301	$0\ .\ 111\ldots \times 2^{-2}$
	v_3	$0.359081397034\ldots$	0x5BECC229	$.\ 1110\ldots \times 2^{-4}$

Table 15.6: Coefficients of the minimax polynomial in (15.3) approximating $\log_2(m_x)/(m_x - 1)$.

Unlike for sine and cosine, both FDlibm and FLIP 0.3 perform range reduction step here. The large difference in the number of cycles is chiefly due to the function calls to perform floating-point operations in FDlibm.

Library	2^x	$\log_2(x)$
FDlibm + Original	726	1456
FDlibm + FLIP 0.2	500	1009
FLIP 0.3	38	47

Table 15.7: Performances of the exponential and the logarithm functions as measured on ST231.

The improvements for the second way with respect to the first way of measuring the performances can be given in terms of speed-up factors which are 1.45 and 1.44 for 2^x and $\log_2(x)$ respectively. Again, the speed-up factors for both functions lie between the speed-up factors of addition/subtraction, multiplication and FMA/FMS. We find that if FDlibm is used, FLIP 0.2 is preferable to the original library.

Conclusion

This thesis provides a software implementation of floating-point algorithms optimized for two integer processors, ST220 and ST231 form STMicroelectronics. We have addressed the design, test and implementation issues involved in performing single precision floating-point arithmetic on these processors. This work did not simply map the available software implementations to the target processors; instead it came up with modified software implementations and some specific algorithms which have been well adapted to the target processors. Hence, different possibilities of optimization both at the architectural level and at the algorithmic level have been studied. In all, twelve floating-point operations have been developed. During the implementation of the algorithms we relied on the IEEE floating-point standard. Although we are not fully compatible with it, we still provide correctly rounded results both for normalized and subnormal numbers for basic arithmetic operations. Due to speed constraints other operations only support normalized numbers and one rounding mode (round to nearest). The software library FLIP is a concrete realization this work.

FLIP provides further optimizations to an already optimized library from STMicroelectronics for existing operations. Also, it provides support for additional operations and elementary functions optimized for our target processors.

Contributions

Avoiding Comparisons

In general, comparisons are detrimental to parallelism. The target processors allow to execute four instructions encoded in a bundle in one cycle. In order to get an optimum bundle fill rate, the compiler must extract the parallelism from the program or the programmer must help compiler by writing programs that performs computation in a parallel way and try to avoid/replace some costly comparisons. For handling special values four comparisons are replaced with a single comparison. Normally, post-normalization requires detection of a particular bit pattern of the result *after* rounding. This detection has been replaced by an addition and a shift and is performed *before* rounding. In the radix-4 SRT algorithm, three comparisons for choosing the selection constants are replaced by a comparison-free alternative. Due to this contribution FLIP is able to perform operations such as $\pm$ and $\times$ 1.2 times (at least) faster in comparison to the original library.

IEEE Compatibility

For $\pm, \times, /, \sqrt{}$ subnormal numbers and all the four rounding modes can be supported. Since the rounding mode chosen by the user is set *dynamically* through a global variable, branches due to comparisons with this variable are induced at several points during execution. This dynamic support offsets almost all the performance gained through optimizations done inside the code. When the rounding modes are supported *statically* then, contrary to the belief, subnormal numbers

and all the rounding modes can be supported with an insignificant increase in latency (5-9%) and code size (0.8%).

Polynomial Evaluation

Different methods for evaluating polynomials have been tried. We have modified the Estrin method to evaluate polynomials on our target processors more *efficiently* and as *accurately* possible. It has been found out that polynomials of higher degree (≥ 4) should not be evaluated on ST220 due to its less accurate rectangular multiplier. Moreover, when a full-width multiplier is available as in the case of ST231, one should prefer a polynomial of odd degree to a polynomial of even degree. Our method of evaluation performs the computations in a highly parallel fashion. Nevertheless, it tries to balance the number of additions and multiplications performed during evaluation with the loss of accuracy due to evaluation errors.

Improvement of Division and Square Root

These operations are generally regarded as the slowest among the basic operations. To accelerate division and square root we have tried different types of algorithms and finally came up with an implementation based on a *specific* algorithm for a *particular* processor: we recommend very-high radix SRT algorithm for ST220 and Goldschmidt algorithm for ST231.

The FLIP Library

We have developed the FLIP library supporting twelve floating-point operations to be performed efficiently (both in time and code size) on ST220 and ST231. With FLIP, we have obtained better performances in comparison to the original library which has been used as the reference library. The number of lines of code written for each version of FLIP and for its validation is given below

	FLIP 0.1	FLIP 0.2	FLIP 0.3
FLIP	1566	4574	5338
Validation	1263	2315	3195

FLIP allowed to obtain the following speed-up factors in comparison to the original library. The speed-up factor for both libraries has been computed based on the timings as measured on ST231.

Operations	$+/-$	$\times$	$/$	$\sqrt{x}$	x^2	FMA/FMS	$1/x$	$1/\sqrt{x}$
Speedup factor	1.38/1.37	1.25	3.47	2.49	1.66	1.79/1.78	3.76	4.53

Finally, a rewarding achievement for FLIP is that, in September 2005, it has replaced the original STMicroelectronics library in the industrial compiler of the ST200 family.

Future Prospects

Cheap Support for all Rounding Modes

Since the dynamic support for rounding modes is very expensive because of some comparisons, we will try to find out first whether any of the comparisons can be suppressed or replaced by some trivial operations (additions, shifts). Next, we will ask STMicroelectronics whether some specific compiler optimizations can be done in order to suppress the comparisons with the global variable.

Support for other Operations

We will try to enrich FLIP with some other operations like $1/\sqrt{x^2+y^2}$, x^3, $(\sin, \cos)$. These operations are used in various applications. We will restrict the support, at least in the first instant, to normalized numbers and round to nearest even mode.

Porting FLIP to other Processors

Concerning other processors, the first step will be to choose a processor with similar characteristics to ST220 and ST231. Next, FLIP without any modifications will be measured on this processor. There might be a significant difference in the performance of FLIP. The advantages and drawbacks of the processor will be studied in order to know what kind of optimizations are required to boost the performance of FLIP.

FLIP for Double Precision

In order to extend the library for double precision, the foremost thing is to know which operations can be reused by slightly modifying its implementation at the architectural level. Fast implementation of some operations might require changes at the algorithmic level. A complete new battery of tests will have to be written for validation of the library and, today, exhaustive validation of even the unary operations is still "impossible". Nevertheless, all the cases of special values can still be checked.

Appendix A

Proof of Polynomial Evaluation through Gappa

Here we give an overview of how Gappa [65] from G. Melquiond helps us to prove that the polynomial in (9.15) with its coefficients given in Table 9.4 gives an approximation $\mathcal{F}$ of $1/(1+f_y)$, $0.404 \leq f_y < 1$ ($m_y = 1.f_y$) such that

$$1 - 2^{-9} \leq Z = \mathcal{F} \times m_y \leq 1 + 2^{-9}.$$

We give a sketch of Gappa proof with the corresponding mathematical expressions and its implementation in C. In evaluation of the polynomial we use truncated multiplications so we have to tell Gappa the precision of each operand and also the precision in which to produce the result of each intermediate operation. The truncated values TRUNCf_y and TRUNCT2 are formed after chopping off all the bits after the 14th fractional bit as specified by `<fixed,-14>`. The infinitely precise coefficients have been truncated and their hexadecimal representation is used for computation. The values used here are u_2=0x78F57C05 and $u_3 = 0.125$. Note that the value $u_3 f_y$ is computed by shifting f_y right by 3 bits and is an exact operation.

Evaluation of a part of the polynomial.	C code showing the evaluation.	Gappa sketch of the evaluation.
		# 0.12345678901234567890123456789012 # 0.xxxxxxxxxxxxxxxxxxxxxxxx00000000 f_y = <fixed,−23>(Mf_y); #(1, 23)
	yFrac_16H = yFrac >> 16;	# 0.12345678901234567890123456789012 # 0.xxxxxxxxxxxxxx0000000000000000 TRUNCf_y = <fixed,−14>(f_y); #(1,14)
$f_y^2 = f_y \times f_y$	yFracSqr = __st220mulhhs(yFrac, yFrac_16H);	# 0.12345678901234567890123456789012 # 0.xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx f_y2 <fixed,−31>= *trunc*(f_y) * f_y; #(1,31)
$u_3 f_y = 0.125 \times f_y$	u3yFrac = yFrac >> 3;	# 0.12345678901234567890123456789012 # 0.000xxxxxxxxxxxxxxxxxxxxxxx00000 u3f_y = u3 * f_y; #(1,26), exact operation
T2 = $u_2 - u_3 f_y$	T2 = (u2 − u3yFrac);	# 0.12345678901234567890123456789012 # 0.xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx T2 = u2 − u3f_y; #(1,31)
	T2_16H = T2 >> 16;	# 0.12345678901234567890123456789012 # 0.xxxxxxxxxxxxxx0000000000000000 TRUNCT2 = <fixed,−14>(T2); #(1,14)
T3 = $T2\ f_y^2$	T3 = __st220mulhhs(yFracSqr, T2_16H);	# 0.12345678901234567890123456789012 # 0.xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx T3 <fixed,−31>= f_y^2 * TRUNCT2; #(1,31)

Figure A.1: Evaluation of a degree-3 polynomial, its implementation in C and proof of its implementation through Gappa for the interval $[0.404, 1)$.

Bibliography

[1] CRlibm.
URL: *http://lipforge.ens-lyon.fr/www/crlibm/index.html*.

[2] DGL the data generation language.
URL: *http://cs.ecs.baylor.edu/∼maurer/*.

[3] European telecommunications standards institute–ETSI.
URL: http://www.etsi.org.

[4] FDlibm.
URL: *ftp://sunsite.unc.edu/pub/packages/development/libraries/*.

[5] GCC, a GNU Compilation Ccollection.
URL: *http://www.gnu.org/software/gcc/gcc.html*.

[6] IEEE 754 revision work.
URL: *http://grouper.ieee.org/groups/754/revision.html*.

[7] International telecommunication union–ITU.
URL: http://www.itu.int.

[8] libmcr.
URL: *http://www.sun.com/download/products.xml?id=41797765*.

[9] Linux Programmers Manual.
URL: *http://www.tldp.org/guides.html*.

[10] Multiflow computer, VLIW, history, information.
URL: http://www.reservoir.com/vliw.php.

[11] Open64 compiler tools.
URL: *http://open64.sourceforge.net/*.

[12] Intel Pentium processor data book. Intel Corporation, 1994.

[13] R. C. Agarwal, J. C. Cooley, F. G. Gustavson, J. B. Shearer, G. Slishman, and B. Tuckerman. New scalar and vector elementary functions for the IBM system/370. *IBM Journal of Research and Development*, 30(2):126–144, March 1986.

[14] American National Standards Institute and Institute of Electrical and Electronics Engineers. IEEE standard for binary floating-point arithmetic. *ANSI/IEEE Standard, Std 754-1985, New York*, 1985.

[15] American National Standards Institute and Institute of Electrical and Electronics Engineers. IEEE standard for radix independent floating-point arithmetic. *ANSI/IEEE Standard, Std 854-1987, New York*, 1987.

[16] D. E. Atkins. High-radix division using estimates of the divisor and partial remainders. *IEEE Transactions on Computers*, 17(10):925–934, October 1968.

[17] A. Beaumont-Smith, N. Burgess, S. Lefrere, and C. Lim. Reduced latency IEEE floating-point standard adder architectures. In I. Koren and P. Kornerup, editors, *Proceedings of 14th IEEE Symposium on Computer Arithmetic*, pages 35–43, Adelaide, Australia, April 1999.

[18] C. Bertin, N. Brisebarre, B. Dupont de Dinechin, C.-P. Jeannerod, C. Monat, J.-M. Muller, S. K. Raina, and A. Tisserand. A floating-point library for integer processors. In Franklin T. Luk, editor, *Proceedings of SPIE, Advanced Signal Processing Algorithms, Architectures, and Implementations XIV*, volume 5559-11, pages 101–111, Denver, Colorado, USA, August 2004.

[19] J. Cocke and D. W. Sweeney. High speed arithmetic in a parallel device. Technical report, IBM, 1957.

[20] W. Cody and W. Waite. *Software Manual for the Elementary Functions*. Prentice-Hall, 1980.

[21] T. Coe and P. T. P. Tang. It takes six ones to reach a flaw. In S. Knowles and W. H. McAllister, editors, *Proceedings of 12th IEEE Symposium on Computer Arithmetic*, pages 140–146, Bath, England, UK, July 1995.

[22] M. A. Cornea-Hasegan, R. A. Golliver, and P. Markstein. Correctness proofs outline for Newton-Raphson based floating-point divide and square root algorithms. In I. Koren and P. Kornerup, editors, *Proceedings of 14th IEEE Symposium on Computer Arithmetic*, pages 96–105, Adelaide, Australia, April 1999.

[23] F. de Bont, M. Groenewegen, and W. Oomen. A high-quality audio-coding system at 128kb/s. In *98th AES Convention*, pages 25–28, February 1995.

[24] B. Dupont de Dinechin, F. de Ferrière, C. Guillon, and A. Stoutchinin. Code generator optimizations for the ST120 DSP-MCU core. In *Proceedings of the 2000 International Conference on Compilers, architecture, and synthesis for embedded systems*, pages 93–102, San Jose, California, USA, November 2000. ACM Press, New York, USA.

[25] F. de Dinechin and A. Tisserand. Some improvements on multipartite table methods. In N. Burgess and L. Ciminiera, editors, *Proceedings of 15th IEEE Symposium on Computer Arithmetic*, pages 128–135, Vail, Colorado, USA, June 2001.

[26] D. Defour. *Fonctions Elémentaires : algorithmes et implémentations efficaces pour l'arrondi correct en double précision.* PhD thesis, Ecole Normale Supérieure de Lyon, September 2003.

[27] T. J. Dekker. A floating point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, June 1971.

[28] A. Edelman. The mathematics of the Pentium division bug. *SIAM Review*, 39(1):54–67, March 1997.

[29] M. D. Ercegovac. A higher radix division with simple selection of quotient digits. In T. R. N. Rao and P. Kornerup, editors, *Proceedings of 6th IEEE Symposium on Computer Arithmetic*, pages 94–98, Aarhus University, Aarhus, Denmark, June 1983.

[30] M. D. Ercegovac, L. Imbert, D. W. Matula, J.-M. Muller, and G. Wei. Improving Goldschmidt division, square root, and square root reciprocal. *IEEE Transactions on Computers*, 49(7):759–766, July 2000.

[31] M. D. Ercegovac and T. Lang. Simple radix-4 division with operands prescaling. *IEEE Transactions on Computers*, 39(9):1204–1208, September 1990.

[32] M. D. Ercegovac and T. Lang. *Division and Square-Root Algorithms: Digit-Recurrence Algorithms and Implementations*. Kluwer Academic, 1994.

[33] M. D. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kaufmann, 2003.

[34] M. D. Ercegovac, T. Lang, and P. Montuschi. Very-high radix division with prescaling and selection by rounding. *IEEE Transactions on Computers*, 43(8):909–918, August 1994.

[35] J. F. Hart et al. *Computer Approximations*. Wiley, 1968.

[36] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. M. O. Homewood. Lx: a technology platform for customizable VLIW embedded processing. In *Proceedings of of the 27th International Symposium on Computer Architecture*, pages 203–213, Vancouver, British Columbia, Canada, June 2000. ACM Press, New York, USA.

[37] P. Farmwald. High bandwidth evaluation of elementary functions. In K. S. Trivedi, editor, *Proceedings of 5th IEEE Symposium on Computer Arithmetic*, pages 139–142, Ann Arbor, Michigan, May 1981.

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

[39] W. Ferguson and T. Brightman. Accurate and monotone approximations of some transcendental functions. In P. Kornerup and D. W. Matula, editors, *Proceedings of 10th IEEE Symposium on Computer Arithmetic*, pages 237–234, Grenoble, France, June 1991.

[40] C. Finot-Moreau. *Preuves et algorithmes utilisant l'arithmétique flottante normalisée IEEE*. PhD thesis, Ecole Normale Supérieure de Lyon, July 2001.

[41] M. J. Flynn. On division by functional iteration. *IEEE Transactions on Computers*, C-19:194–198, August 1970. Reprinted in Computer Arithmetic, E. E. Swartzlander, vol. 1. Los Alamitos, California: IEEE Computer Society Press, 1990.

[42] S. Gal and B. Bachelis. An accurate elementary mathematical library for the IEEE floating-point standard. *ACM Transactions on Mathematical Software*, 17(1):26–45, March 1991.

[43] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, March 1991.

[44] R. E. Goldschmidt. Applications of division by convergence. Master's thesis, Massachusetts Institute of Technology, June 1964. MSc dissertation.

[45] F. G. Gustavson, J. E. Moreira, and R. F. Enenkel. The fused multiply-add instruction leads to algorithms for extended-precision floating point: applications to java and high-performance computing. In S. A. Mackay and J. H. Johnson, editors, *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative research*, page 4, Mississauga, Ontario, Canada, November 1999. IBM Press.

[46] H. Hassler and N. Takagi. Function evaluation by table lookup and addition. In S. Knowles and W. H. McAllister, editors, *Proceedings of 12th IEEE Symposium on Computer Arithmetic*, pages 10–16, Bath, England, UK, July 1995.

[47] J. Hauser. SoftFloat.
URL: *http://www.jhauser.us/arithmetic/SoftFloat.html*.

[48] J. Hauser. TestFloat.
URL: *http://www.jhauser.us/arithmetic/TestFloat.html*.

[49] G. L. Haviland and A. A. Tuszynski. A CORDIC arithmetic processor. *IEEE Transactions on Computers*, C-44(2):68–79, 1980.

[50] J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann, third edition, 2003.

[51] E. Hokenek, R. Montoye, and P. W. Cook. Second-generation RISC Floating Point with Multiply-Add Fused. *IEEE Journal of Solid-State Circuits*, 25(5):1207–1213, October 1990.

[52] D. Hough. Elementary functions based on IEEE arithmetic. In *Mini/Micro West Conference Record*, pages 1–4. Electronic Conventions Inc., 1983.

[53] HP and STMicroelectronics. LX architecture.
URL: *http://www.embedded.com/2000/0010/0010feat6.htm*.

[54] S. F. Hsiao and J. M. Delosme. Householder CORDIC algorithms. *IEEE Transactions on Computers*, C-44:990–1000, 1995.

[55] Maple Inc. Maple.
URL: *http://www.maplesoft.com/*.

[56] C. Iordache and D. W. Matula. On infinitely precise rounding for division, square root, reciprocal and square root reciprocal. In I. Koren and P. Kornerup, editors, *Proceedings of 14th IEEE Symposium on Computer Arithmetic*, pages 233–240, Adelaide, Australia, April 1999.

[57] C.-P. Jeannerod, S. K. Raina, and A. Tisserand. High-radix floating-point division algorithms for embedded VLIW integer processors. In *17th IMACS World Congress Scientific Computation, Applied Mathematics and Simulation*, July 2005.

[58] W. Kahan. A logarithm too clever by half. Available at URL: *http://http.cs.berkley.edu/ wkahan/LOG10HAF.TXT*, 2004.

[59] D. Knuth. *The Art of Computer Programming, "Seminumerical Algorithms"*, volume 2. Addison-Wesley, Reading, MA, USA, third edition, 1998.

[60] E. V. Krishnamurthy. On range-transformation techniques for division. *IEEE Transactions on Computers*, C-19(2):157–160, February 1970.

[61] V. Lefèvre. *Moyens arithmétiques pour un calcul fiable*. PhD thesis, Ecole Normale Supérieure de Lyon, January 2000.

[62] T. Lynch, A. Ahmed, M. J. Schulte, T. Callaway, and R. Tisdale. The K5 transcendental functions. In S. Knowles and W. H. McAllister, editors, *Proceedings of 12th IEEE Symposium on Computer Arithmetic*, pages 222–229, Bath, England, UK, July 1995.

[63] P. Markstein. Computation of elementary functions on the IBM RISC system/6000 processor. *IBM Journal of Research and Development*, 34(1):111–119, January 1990.

[64] P. Markstein. Software division and square root using Goldschmidt algorithm. In V. Brattka and N. Muller, editors, *6th Conference on Real Numbers and Computers*, pages 146–157, Dagstuhl, Germany, November 2004.

[65] G. Melquiond. GAPPA, a tool for automatic proof generation of arithmetic properties. URL: *http://lipforge.ens-lyon.fr/www/gappa/*.

[66] J.-M. Muller. A few results on table-based methods. *Reliable Computing*, 5(3):279–288, 1999.

[67] J.-M. Muller. On the definition of ulp(x). Technical Report RR2005-09, LIP, ENS Lyon, France, February 2005.

[68] J.-M. Muller. *Elementary Functions: Algorithms and Implementation*. Birkhauser Boston, 2nd edition, 2006.

[69] S. F. Oberman and M. J. Flynn. Division algorithms and implementations. *IEEE Transactions on Computers*, 46(8):833–854, August 1997.

[70] M. L. Overton. *Numerical Computing with IEEE Floating Point Arithmetic*. Society for Industrial and Applied Mathematics, 2001.

[71] W.-C. Park, T.-D. Han, S.-D. Kim, and S.-B. Yang. Floating-point adder/subtractor performing IEEE rounding and addition/subtraction in parallel. *IEICE Transaction on Information and Systems*, E79-D(4):297–305, April 1996.

[72] M. Parks. Number-theoretic test generation for directed roundings. In I. Koren and P. Kornerup, editors, *Proceedings of 14th IEEE Symposium on Computer Arithmetic*, pages 241–248, Adelaide, Australia, April 1999.

[73] N. Quach, N. Takagi, and M. Flynn. On fast IEEE rounding. Technical Report CSL-TR-91-459, Stanford University, January 1991.

[74] C. V. Ramamoorthy, J. R. Goodman, and K. H. Kim. Some properties of iterative square-rooting methods using high-speed multiplication. *IEEE Transactions on Computers*, C-21:837–847, August 1972.

[75] E. Remez. Sur un procédé convergent d'approximations successives pour déterminer les polynômes d'approximation. *C.R. Académie des Sciences, Paris*, 198, 1934.

[76] S. E. Richardson. Exploiting trivial and redundant computation. In I. Koren and P. Kornerup, editors, *Proceedings of 11th IEEE Symposium on Computer Arithmetic*, pages 220–227, Windsor, Ontario, Canada, July 1993.

[77] J. E. Robertson. A new class of digital division methods. *IEEE Transactions on Electronic Computers*, EC-7:218–222, September 1958.

[78] D. D. Sarma and D. W. Matula. Faithful bipartite ROM reciprocal tables. In S. Knowles and W. H. McAllister, editors, *Proceedings of 12th IEEE Symposium on Computer Arithmetic*, pages 17–28, Bath, England, UK, July 1995.

[79] M. Schulte and E. E. Swartzlander. Exact rounding of certain elementary functions. In E. E. Swartzlander, M. J. Irwin, and G. Jullien, editors, *Proceedings of 11th IEEE Symposium on Computer Arithmetic*, pages 138–145, Windsor, Canada, June 1993.

[80] M. J. Schulte and Jr. E. E. Swartzlander. Hardware designs for exactly rounded elementary functions. *IEEE Transactions on Computer, Special Issue on Computer Arithmetic*, 43(8):964–973, August 1994.

[81] P.-M. Seidel. *On the Design of IEEE Compliant Floating-Point Units and Their Quantitative Analysis*. PhD thesis, University of Saarland, Germany, December 1999.

[82] P.-M. Seidel and G. Even. How many logic levels does floating-point addition require. In *Proceedings of International Conference Computer Design: VLSI in Computers and Processors*, pages 142–149, Austin, Texas, USA, October 1998. IEEE Computer Society Press.

[83] P.-M. Seidel and G. Even. On the design of fast IEEE floating-point adders. In N. Burgess and L. Ciminiera, editors, *Proceedings of 15th IEEE Symposium on Computer Arithmetic*, pages 184–194, Vail, Colorado, USA, June 2001.

[84] P.-M. Seidel and G. Even. Delay-optimized implementation of IEEE floating-point addition. *IEEE Transactions on Computers*, 53(2):97–113, February 2004.

[85] H. Sharangpani and K. Arora. Itanium Processor Microarchitecture. In *IEEE Micro Magazine*, volume 20, pages 24–43. September 2000.

[86] J. D. Silverstein, S. E. Sommars, and Y. C. Tao. The UNIX system math library, a status report. Technical report, USENIX—Winter'90, 1990.

[87] J. E. Stine and M. J. Schulte. The symmetric table addition method for accurate function. *VLSI Signal Processing*, 21(2):167–177, 1999.

[88] STMicroelectronics. *ST200 Core and Instruction Set Architecture Manual*, October 2002. URL: *http://www.st.com*.

[89] STMicroelectronics. *ST200 Cross Development Manual*, November 2004. URL: *http://www.st.com*.

[90] STMicroelectronics. *ST200 Micro Toolset User Manual*, November 2004. URL: *http://www.st.com*.

[91] A. Svoboda. An algorithm for division. *Information Processing Machines (Stroje na Zpracovani Informaci)*, 9:25–34, 1963.

[92] K. G. Tan. The theory and implementation of high-radix division. In A. Avizienis M. D. Ercegovac, editor, *Proceedings of 4th IEEE Symposium on Computer Arithmetic*, pages 154–163, Santa Monica, California, USA, October 1978.

[93] K. D. Tocher. Techniques of multiplication and division for automatic binary computers. *The Quarterly Journal of Mechanics and Applied Mathematics*, 11(3):364–384, 1958.

[94] J. E. Volder. The CORDIC trigonometric computing technique. *IRE Transactions on Electronic Computers*, EC-8:330–334, 1959.

[95] J. Walther. A unified algorithm for elementary functions. In *AFIPS Spring Joint Computer Conference*, volume 38, 1971.

[96] S. W. White and S. Dhawan. POWER2: Next generation of the RISC System/6000 family. *IBM Journal of Research and Development*, 38(5):493–502, September 1994.

[97] J. J. Yi and D. J. Lilja. Improving processor performance by simplifying and bypassing trivial computation. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 462–466, 2002.

Résumé

Aujourd'hui, les systèmes embarqués sont omniprésents dans de nombreuses applications. À cause de la vitesse et de la consommation d'énergie la plupart des processeurs dans les systèmes embarqués ont seulement des ressources matérielles de calcul entier (ou virgule fixe). Un grand nombre des applications en calcul scientifique et en traitement du signal utilisent l'arithmétique virgule flottante pour des raisons de précision. Donc, pour faire du calcul numérique sur les nombres flottants, soit on utilise des processeurs avec une couche flottante matérielle qui consomme beaucoup d'énergie, soit on fait une émulation logicielle de la couche flottante pour les processeurs entiers.

Le travail de cette thèse concerne la conception et l'implantation de couches logicielles pour l'arithmétique virgule flottante sur les processeurs de la famille ST200 de STMicroelectronics. Le but de cette thèse, réalisée dans le cadre d'une collaboration avec STMicroelectronics et financée par la Région Rhône-Alpes, est de concevoir une bibliothèque pour les processeurs de la famille ST200. Ce travail présente les possibilités d'adéquation et d'adaptation des algorithmes spécifiques à ces processeurs. Il montre qu'une amélioration considérable de la performance peut être obtenue grâce à une bonne connaissance de l'architecture cible, un choix judicieux de l'algorithme à utiliser et un réglage minutieux de l'implantation. La bibliothèque FLIP est la concrétisation logicielle de cette thématique. Elle implante les cinq opérations de base $(+, -, \times, /, \sqrt{})$ en respectant la norme IEEE-754, des opérations supplémentaires comme x^2, $xy \pm z$, $1/x$, $1/\sqrt{x}$ avec une implantation rapide de la division et de la racine carrée, et quelques fonctions élémentaires ($\sin(x)$, $\cos(x)$, $\log_2 x$, 2^x). FLIP a été comparée à la bibliothèque initiale déjà optimisée par STMicroelectronics; pour chaque opération FLIP a obtenu de meilleurs performances. Des gains d'au moins 40% pour deux applications industrielles ont permis à FLIP de remplacer la bibliothèque initiale de STMicroelectronics dans la version R4.1 des compilateurs pour ST200.

Mots-clés : arithmétique flottante, processeurs entiers, addition, multiplication, division et racine carrée, multiplication addition fusionnées, fonctions élémentaires.

Abstract

Embedded systems are now ubiquitous in a wide range of diverse application domains. Because of cost and energy reasons most embedded systems rely on *integer* or *fixed-point processors*. Such systems confront with a problem while supporting the multimedia applications which depend on floating-point arithmetic due to high demands for accuracy. A straightaway solution might be to use a dedicated floating-point unit but it comes at the cost of extra silicon area and increased power consumption. The solutions existing in software consist of introducing some scaling operations in the target program, which becomes complicated due to wide range of floating-point numbers, or porting the programs meant for general-purpose processors to integer or fixed-point cores which is quite a complex task. In those cases, an efficient solution may be to use a fast floating-point emulation layer over the integer or fixed-point units.

This thesis addresses the design and implementation issues involved in providing this *software layer* to support floating-point arithmetic on the ST200 family of VLIW integer processors from STMicroelectronics. This work discusses the possibilities of adequation and adaptation of specific algorithms to ST200 processors. It shows that the performance of a software implementation can be significantly improved through a sound understanding of a particular architecture, careful selection of an algorithm and a fine tuning of an implementation. FLIP, developed in a collaborative project funded by Région Rhône-Alpes and STMicroelectronics, supports basic floating-point operations $(+, -, \times, /, \sqrt{})$, additional operations $(x^2, xy \pm z, 1/x, 1/\sqrt{x})$ with a fast implementation of division and square root and elementary functions $(\exp(x), \log_2(x), \sin(x), \cos(x))$. FLIP has achieved better performances in comparison to the original library which has already been optimized by STMicroelectronics. For complex operations like division and square root, FLIP is at least three times faster.

Keywords: floating-point arithmetic, integer processors, floating-point addition, multiplication, division, square root, fused multiply-and-add, elementary functions.