Techniques and tools for implementing IEEE 754 floating-point arithmetic on VLIW integer processors

Claude-Pierre Jeannerod,[*] Christophe Mouilleron,[†] Jean-Michel Muller,[‡] Guillaume Revy[§]
Laboratoire LIP (CNRS, ENS de Lyon, INRIA, UCBL), Université de Lyon, France
Email: firstname.lastname@ens-lyon.fr

Christian Bertin, Jingyan Jourdan-Lu,[¶] Hervé Knochel, Christophe Monat,
STMicroelectronics
Compilation Expertise Center, Grenoble, France
Email: firstname.lastname@st.com

ABSTRACT

Recently, some high-performance IEEE 754 single precision floating-point software has been designed, which aims at best exploiting some features (integer arithmetic, parallelism) of the STMicroelectronics ST200 Very Long Instruction Word (VLIW) processor. We review here the techniques and software tools used or developed for this design and its implementation, and how they allowed very high instruction-level parallelism (ILP) exposure. Those key points include a hierarchical description of function evaluation algorithms, the exploitation of the standard encoding of floating-point data, the automatic generation of fast and accurate polynomial evaluation schemes, and some compiler optimizations.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*standards*; G.1.0 [**Numerical Analysis**]: General—*computer arithmetic, parallel algorithms*; G.4 [**Mathematical Software**]: *algorithm design and analysis, parallel and vector implementations*; D.3.4 [**Programming Languages**]: Processors—*code generation, compilers*

General Terms

Algorithms, Design, Performance, Reliability

Keywords

binary floating-point arithmetic, correct rounding, IEEE 754, polynomial evaluation, instruction-level parallelism, C software implementation, code generation, VLIW processor

[*]INRIA.
[†]ENS de Lyon.
[‡]CNRS.
[§]With LIP and ENS de Lyon when this work has been done, but now a member of ParLab within EECS Department at the University of California at Berkeley. E-mail: grevy@eecs.berkeley.edu
[¶]Also a member of LIP. Email: lujingyan@gmail.com

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PASCO 2010, 21–23 July 2010, Grenoble, France.
Copyright 2010 ACM 978-1-4503-0067-4/10/0007 ...$10.00.

1. INTRODUCTION

Although binary floating-point arithmetic has been standardized since over two decades and is widely used in media-intensive applications, not all embedded media processors have floating-point hardware. An example is the ST231, a 4-way VLIW processor from the STMicroelectronics ST200 family whose native arithmetic consists exclusively of 32-bit integer arithmetic. Consequently, for such so-called *integer* processors floating-point arithmetic must be implemented entirely in software.

Various software implementations of the IEEE 754 standard [1] for binary floating-point arithmetic already exist, such as the SoftFloat package [16]. SoftFloat is written in portable C and could thus be compiled for the ST231 using the ST200 C compiler. While being entirely satisfactory in terms of correctness, such a reference library may fail to exploit some key features of the given target, thus offering standard floating-point support at a possibly prohibitive cost. For example, for the five basic arithmetic operations (addition, subtraction, multiplication, division, and square root) in single precision and with rounding 'to nearest even', we get the following latencies (measured in number of clock cycles):

$+$	$-$	$\times$	$/$	$\sqrt{}$
48 cycles	49 cycles	31 cycles	177 cycles	95 cycles

This motivated the study of how to exploit the various features of the ST231, and resulted in the design and implementation of improved C software for floating-point functionalities gathered in a library called FLIP [14].[1] Compared

[1]The earliest report on this study is [4] and the latest release, FLIP 1.0, is available from `http://flip.gforge.inria.fr/` and delivered under the CeCILL-v2 license.

RIGHTSLINK

to SoftFloat, this alternative implementation of IEEE 754 arithmetic allows to obtain speed-ups from 1.5x to 5.2x, the new latencies being as follows (see [38, p. 4]):

$+$	$-$	$\times$	$/$	$\sqrt{}$
26 cycles	26 cycles	21 cycles	34 cycles	23 cycles

This study has shown further that similar speed-ups hold not only for rounding to nearest but in fact for all rounding modes and, perhaps more interestingly, that supporting so-called *subnormal numbers* (the tiny floating-point numbers that make gradual underflow possible) has an extra-cost of only a few cycles. Also, square root is now almost as fast as multiplication, and even slightly faster than addition. Finally, the code size (number of assembly instructions) has been reduced by a factor ranging between 1.2 and 4.2 depending on the operator.

To achieve such performances required the combination of various techniques and tools. The main techniques used are a careful exploitation of some nice features of the IEEE standard, a novel algorithmic approach to increase ILP exposure in function evaluation, as well as some compiler optimizations. The software tools used to assist the design and its validation are Gappa, Sollya, and CGPE. The goal of this paper is to review such techniques and tools through a few examples, showing how they can help expose more ILP while keeping code size small and allowing for the validation of the resulting codes. Although the optimizations we present here have been done for the ST231, most of them could still be useful when implementing IEEE floating-point arithmetic on other VLIW integer processors.

The paper is organized as follows. Section 2 provides some reminders about the IEEE 754 floating-point standard (Section 2.1), the architecture of the ST231 processor (Section 2.2), and the associated compiler (Section 2.3). In Section 3 we provide a high-level description of typical floating-point arithmetic implementations, which already allows to expose a great deal of ILP and to identify the most critical subtasks. We then detail three optimization examples: Section 4 presents two ways of taking advantage of the encoding of floating-point data prescribed by the IEEE 754 standard; Section 5 shows how some automatically-generated parallel polynomial evaluation schemes allow for even higher ILP exposure and can be used to accelerate the most critical part of the implementation; Section 6 details an example of optimization done at the compiler level. We conclude in Section 7 with some remarks on the validation of the optimized codes thus produced.

2. BACKGROUND

2.1 IEEE 754 floating-point arithmetic

Floating-point arithmetic is defined rigorously by the IEEE 754 standard [1, 2], whose initial motivation was to make it possible to "write portable software and prove that it worked" [25]. We simply recall some important features of this standard and refer to [6, 17, 34, 35] for in-depth descriptions.

The *data* specified by this standard consist of finite numbers, signed infinities, and quiet or signaling Not-a-Numbers (NaNs). This standard also defines several *formats* characterized by the values of three integers: the radix β, the precision p, and the maximal exponent $e_{\max}$. Although β can be either 2 or 10, we shall restrict here to binary formats, for which $\beta = 2$ and where $e_{\max}$ has the form

$$e_{\max} = 2^{w-1} - 1$$

for some positive integer w; in our implementation examples we shall restrict further to the *binary32* format (formerly called *single precision*), for which $p = 24$ and $e_{\max} = 127 = 2^7 - 1$.

For a given format, finite numbers have the form

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

where

- $s_x \in \{0, 1\}$;
- $e_x \in \{e_{\min}, \dots, e_{\max}\}$ with $e_{\min} = 1 - e_{\max}$;
- $m_x = M_x \cdot 2^{1-p}$ with M_x an integer such that $0 \leq M_x < 2^p$ if $e_x = e_{\min}$, and $2^{p-1} \leq M_x < 2^p$ if $e_x > e_{\min}$.

The number x in (1) is called *subnormal* if $0 < |x| < 2^{e_{\min}}$, and *normal* if $|x| \geq 2^{e_{\min}}$. In particular, one may check that x is subnormal if and only if $e_x = e_{\min}$ and $1 \leq M_x < 2^{p-1}$. Subnormals are a key feature of the IEEE 754 standard, as they allow for gradual rather than abrupt underflow [26]. Another important feature of the above definition of x is that it gives two signed zeros, $+0$ and -0.

Every floating-point datum x can be represented by its standard *encoding* into a k-bit integer

$$X = \sum_{i=0}^{k-1} X_i 2^i, \tag{2}$$

where $k = p + w$. When x is a finite number, the bits X_i of X must be interpreted as follows:

$$[X_{k-1} \cdots X_0] = [\text{sign} | \underbrace{\text{biased exponent}}_{w \text{ bits}} | \underbrace{\text{trailing significand}}_{p-1 \text{ bits}}],$$

where the sign bit equals s_x, the biased exponent equals $e_x + e_{\max}$ for x normal, and 0 for x subnormal, and where the trailing significand contains the fraction bits of m_x. When x is infinite or NaN, special values of X are used, having in particular all the bits of the biased exponent field set to one. In our case, the input and output of the implemented operators correspond to this encoding for $k = 32$. We will see in Section 4 how to take advantage of the nice properties of this encoding to optimize implementations.

Correct rounding is of course another key feature of the IEEE 754 standard: it means that the result to return is the one which would have been produced by first using infinite precision and unbounded exponent range, and then rounding to the target format according to a given rounding direction. The 2008 revision of the standard [2] defines five rounding directions, the default being 'to nearest even.' Correct rounding has been mandatory for the five basic operations since 1985 [1]. Since 2008, this requirement has been extended to several operations and most notably the fused multiply-add (FMA); it is also now recommended that elementary functions (exp, cos, ...) be implemented with correct rounding.

Finally, five *exception flags* must be set after each operation, thus offering a diagnostic of specific behaviors: division by zero, overflow, underflow, inexactness, and non validity of an operation. Although such flags can be very useful when debugging applications, they have not been implemented yet in our context.

RIGHTSLINK
Copyright Clearance Center

2.2 ST200 VLIW architecture

The ST200 is a 4-way VLIW family of cores dedicated to high performance media processing. It was originally designed by Fisher at HPLabs and further developed by STMicroelectronics for use as an IP block in their Systems On Chip (SOCs), such as the 710x family of High Definition TV (HDTV) decoders, where several instances of that core are used for audio and video processing. (We refer to [13] for a comprehensive study of VLIW architectures.)

The flagship processor of this family is the ST231, which has specific architectural features and specific instructions that turn out to be key to support floating-point emulation.

The execution model of the ST200 is the one of a classical VLIW machine where several instructions are grouped in a bundle and execute all at once, with a significant departure from that principle: when a register read-after-write (RAW) dependency is not fulfilled due to the latency of one instruction, the core waits until the result is available for reading. The most important benefit of that feature is that it achieves a very simple form of code compression, avoiding the usage of 'nop' operations that could otherwise be necessary to wait until a result is available.

The ST200 has 64 32-bit general-purpose integer registers and 8 1-bit predicate registers. It has no status flag or similar mechanism: all conditions are computed explicitly either in integer or boolean registers. Boolean registers are then used as predicates for branching instructions, or for specific instructions like addition with carry, or the 'select' instruction that chooses a value among two according to a boolean predicate value. This large set of registers implies that in practice, for the 32-bit code that we consider here, the register pressure is low, and every value can be computed and kept in a register.

The ST200 arithmetic-logic units (ALUs) are replicated for every execution lane available, except for some instructions requiring a relatively large silicon surface to be implemented: for instance four adders are available, meaning that a bundle can execute up to four simultaneous additions, but only two multipliers are implemented, restricting the number of simultaneous multiplications. These ALUs are fully pipelined, and all arithmetic operations have a one cycle latency except the multipliers which have a three cycle latency.

Most ST200 arithmetic instructions can have a 9-bit signed immediate argument, and a specific extension mechanism enables any instruction to have a 32-bit immediate argument by using an extra lane in a bundle. Though this reduces available instruction parallelism, it is an effective mechanism to build large constants, avoiding any access through the data memory, as we will see further.

The two previous features—*two pipelined multipliers* available along with *long immediate operands*—are key for efficient polynomial evaluation; see Section 5.

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

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

The direct mapped organization of the I-cache creates a specific difficulty to obtain good and reproducible performance, since its caching performance is very sensitive to the code layout, due to conflict misses. This has been addressed by post link time optimizations [15] that are either driven by heuristics or by actual code profile.

Note also that in the code presented in this article, the choice was made not to use any in-memory table, to avoid any access through the D-cache side of the system, because a cold miss entails a significant performance hit and the prefetch mechanism cannot be efficient on these types of random accesses.

Finally, we conclude this section with a remark about latency and throughput. For instance, on IA64, Markstein [27] takes care to devise two variants of the emulated floating-point operators, one optimized for latency, one optimized for throughput, that are selected by compiler switches. This makes sense for 'open code generation,' where the operator low-level instructions are emitted directly by the compiler in the instruction stream and not available as library runtime support. This is made possible by the fact that the IA64 has a rich instruction set, that enables emulation of complex operations in a few instructions. This technique enables further optimization and scheduling by the compiler, at the cost of a higher code size. On the contrary for the ST200 where emulation of floating-point operations entails a significant code size, our operators are available only in their library variant, thus optimizing for latency is the criteria of choice.

2.3 ST200 VLIW compiler

The ST200 compiler is based on the Open64 technology,[2] open-sourced by SGI in 2001, and then further developed by STMicroelectronics and more generally by the Open64 community.

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

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

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

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

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

- A dedicated Linear Assembly Optimizer (LAO) is in charge at the CGIR level of software pipelining, pre-

[2] `http://www.open64.net`

RIGHTSLINK

pass and post-pass scheduling. It embeds a nearly optimal scheduler based on an Integer Linear Programming (ILP) formulation of the pipelining problem [3]. As the problem instances are very large, a large neighborhood search heuristic is applied as described in [11] and the ILP problem is further solved by an embedded GLPK (GNU Linear Programming Kit) solver.

- A specific if-conversion phase, designed to transform control flow into 'select' operations [7].
- Some additions to the CGIR 'Extended Block Optimizer' (EBO), including a dedicated 'Range Analysis' and 'Range Propagation' phase.
- A proved and efficient out-of-SSA translation phase, including coalescing improvements [5].

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

Note also that a compiler intrinsic (__builtin_clz) is used to select the specific 'count leading zero' instruction: this is a small restriction to portability since this builtin is supported in any gcc compiler.

3. HIGH LEVEL DESCRIPTIONS

As recalled in Section 2.1, IEEE 754 floating-point data can be either numbers or NaNs, finite or infinite, subnormal or normal, etc. Such a diversity typically entails many particular cases, and considering each of them separately may slow down both the implementation process and the resulting code.

In order to make the implementation process less cumbersome, a first step can be to systematically define which operands should be considered as *special cases*. This means to exhibit a sufficient condition C_{spec} on the input such that the IEEE result belongs to, say, $\{\text{NaN}, \pm\infty, \pm 0\}$. (For NaNs, this condition should be necessary as well.) Then it remains to perform the following three independent tasks:

(T1) Handle *generic cases* (inputs for which C_{spec} is false);

(T2) Handle special cases;

(T3) Evaluate the condition C_{spec}.

The output is produced by selecting the result of either T1 or T2, depending on the result of T3. These three tasks define our highest-level description of an operator implementation. At this level, ILP exposure is clear. To reduce the overall latency, we optimize T1 first, and then only optimize both T2 and T3 (in particular by reusing as much as we can the intermediate quantities used for T1). This is motivated by the fact that task T1 is the one where actual numerical computations, and especially rounding, take place. Thus, for most operators it can be expected that T1 will dominate the costs, and even allow for both T2 and T3 to be performed meanwhile.

The next level of description corresponds to a more detailed view of task T1. Handling generic input typically involves a range reduction step, an evaluation step on the reduced range, and a reconstruction step [33]. For example, for a unary real-valued operator f, the exact value of f at floating-point number x can always be written

$$f(x) = \pm\ell \cdot 2^d,$$

for some real number $\ell \in [1, 2)$ and some integer d. Here ℓ will in general have the form $\ell = F(s, t, \ldots)$, where F is a function either equal to f or closely related to it, and where $s, t, \ldots$ are parameters that encode both range reduction and reconstruction. In our case, s can be a real, non-rational number and t lies in a range smaller than that of x, like $[0, 1]$. Now assume for simplicity that $d \geq e_{\min}$. (The general case can be handled in a similar way at the expense of suitable scalings.) The correctly-rounded result to be returned has the form

$$r = \pm\text{RN}_p(\ell) \cdot 2^d, \qquad (3)$$

where RN_p means rounding 'to nearest even' in precision p.

Task T1 can then be decomposed into three independent sub-tasks: compute the sign s_r of r, the integer d, and the floating-point number $\text{RN}_p(\ell)$. Although $\ell < 2$, its rounded value $\text{RN}_p(\ell)$ can be as large as 2. Again, ILP exposure is explicit at the level within task T1.

For the operators considered here, the most difficult of the three sub-tasks is the computation of $\text{RN}_p(\ell)$. Classically, this sub-task can itself be decomposed into three steps [12], yielding a third level of description: given f and x,

- compute (possibly approximate) values for $s, t, \ldots$;
- deduce from F, s, and t a "good enough" approximation v to ℓ; $\qquad (\star)$
- deduce $\text{RN}_p(\ell)$ be applying to v a suitable correction.

Unlike the two previous levels, this level has steps which are fully sequential. The good news, however, is that the computation of v allows many algorithmic choices, some of them leading to very high ILP exposure; see Section 5.

When the binary expansion $(1.\ell_1\ell_2\cdots)_2$ of ℓ is finite, as is the case for the addition and multiplication operators, correction is done via computing explicitly a rounding bit B such that

$$\text{RN}_p(\ell) = (1.\ell_1 \ldots \ell_{p-1})_2 + B \cdot 2^{1-p}. \qquad (4)$$

When the expansion of ℓ can be infinite, as for square root or division, the situation is more complicated but one can proceed by correcting "one-sided truncated approximations" [12, 20, 21]. With u the truncated value of v after p bits, the correction to apply is now based on whether $u \geq \ell$ or not. For functions like square root or reciprocal, this predicate can be computed *exactly* by means of their inverse functions. (Note however that this kind of decision problem is much more involved for elementary functions (exp, cos, ...) because of the "tablemaker's dilemma" [33].)

The computation of B or the evaluation of $u \geq \ell$ can in general be simplified when the function f to be implemented has properties like

ℓ cannot be exactly halfway between two consecutive floating-point numbers.

Therefore, in order to get simpler and thus potentially faster rounding procedures, a thorough study of the properties of f in floating-point arithmetic can be necessary. Properties like the one above have been derived in [18, 22] for some commonly used algebraic functions.

RIGHTSLINK

4. EXPLOITING STANDARD ENCODINGS

The standard encoding of floating-point data into k-bit integers X as in (2) has several interesting properties, and especially a well-known ordering property (see for example [34, p. 330]). The two examples below show how such properties of the standard encodings of the operand(s) and the result can be exploited to optimize floating-point implementations.

First, the standard encoding can be used to obtain explicit and ready-to-implement formulas for evaluating the condition C_{spec} (task T3). Consider for example the square root operator $x \mapsto \sqrt{x}$. Its IEEE 754 specification implies that

$$C_{\text{spec}} = (x = \pm 0) \ \vee \ (x < 0) \ \vee \ (x = \pm\infty) \ \vee \ (x \text{ is NaN}).$$

A possible implementation of this predicate would thus consist in checking on X if $x = \pm 0$, and so on. However, the following more compact expression was shown in [20]:

$$C_{\text{spec}} = \Big[(X-1) \bmod 2^k \geq 2^{k-1} - 2^{p-1} - 1\Big], \qquad (5)$$

where the notation $[\mathcal{S}]$ means 1 if the statement $\mathcal{S}$ is true, and 0 otherwise. For the binary32 format, this amounts to comparing $(X-1) \bmod 2^{32}$ to the constant value $2^{31} - 2^{23} - 1$. Since on ST231 integer addition is done modulo 2^{32} the above formula can thus be immediately implemented as shown in the C fragment below:

```
Cspec = (X - 1) >= 0x7F7FFFFF;
```

In this case, exploiting the standard encoding allows us to filter out all special cases (task T3) in only 2 cycles and 2 instructions. A similar filter can be designed for addition, multiplication, and division. The overhead due to the fact that these are binary operators is quite reasonable: only 1 more cycle and 3 more instructions are used (see [34, §10]).

As a second example, let us now see how one can exploit the standard encoding at the end of task T1, when packing the result. Let

$$n = RN_p(\ell) = (1.n_1 \dots n_{p-1})_2.$$

For r as in (3), once we have computed its sign s_r as well as n and d, it remains to set up the k-bit integer R that corresponds to the standard encoding of r. One could have concatenated s_r with a biased value

$$D = d + e_{\max}$$

of d and with the fraction bits of n, but removing the leading 1 in n would have increased the critical path. To avoid this, it has been shown in [20] that one may prefer to compute $D-1$ instead of D (at no extra cost since it suffices to modify the value of the bias) and then deduce R as

$$R = \big(s_r \cdot 2^{k-1} + (D-1) \cdot 2^{p-1}\big) + n \cdot 2^{p-1}. \qquad (6)$$

Since the latency of n is in general higher than that of s_r and $D-1$, the evaluation order indicated by the parentheses in (6) may reduce the overall latency of R. Note also that the exponent field is automatically increased by 1 in the case where $n = 2$. In particular, when $n = 2$ and $D - 1 = 2e_{\max} - 1$ then the returned value of R is the encoding of $\pm\infty$, which means that overflow due to rounding is handled transparently thanks to the standard encoding.

Finally, in the cases where n is computed via adding the rounding bit B as in (4), since getting B is the most expensive step, one may rewrite (6) with the following evaluation order:

$$R = \Big(\big(s_r \cdot 2^{k-1} + (D-1) \cdot 2^{p-1}\big) + L\Big) + B, \qquad (7)$$

with L the integer given by $L = (1.\ell_1 \dots \ell_{p-1})_2 \cdot 2^{p-1}$.

5. PARALLEL POLYNOMIAL EVALUATION

When implementing operators like floating-point square root or division, the trickiest part is to write the code computing the approximation v to the real number ℓ; see ($\star$) in Section 3. The goal here is to achieve the lowest possible latency while being "accurate enough" (in a sense made precise in [20], for example $-2^{-p} < \ell - v \leq 0$).

Recall from Section 2.2 that the ST231 can issue up to four integer additions $(A + B) \bmod 2^{32}$ (latency of 1 cycle), up to two multiplications $\lfloor AB/2^{32} \rfloor$ (latency of 3 cycles, pipelined), and that these arithmetic instructions can have 32-bit immediate arguments. These features allow us to consider several methods, such as

- variants of Newton-Raphson and Goldschmidt iterative methods based on low-degree polynomial evaluation followed by 1 or 2 iterations [23, 36];
- methods based on evaluating piecewise, univariate polynomial approximants [19];
- methods based on the evaluation of a single bivariate polynomial [20, 21, 24, 38].

So far the highest ILP exposure and the smallest latencies have been obtained using the third approach: v is obtained by evaluating a very special bivariate polynomial $P(s,t)$ that approximates ℓ "well enough" and has the form

$$P(s,t) = 2^{-p-1} + s \cdot a(t), \qquad a(t) = \sum_{i=0}^{d} a_i t^i.$$

For example, for the square root design introduced in [20], the degree d equals 8 and the coefficients a_i are such that $a_0 = 1$ and, for $1 \leq i \leq 8$, $a_i = (-1)^{i+1} A_i \cdot 2^{-31}$ with A_i a 32-bit positive integer. These numbers as well as a rigorous upper bound on the approximation error entailed by using $P(s,t)$ instead of the true square root function have been produced by the software tool Sollya [9].

Once the polynomial $P(s,t)$ is given, it remains to choose an evaluation scheme that will be fast on ST231 and to bound the rounding errors and check the absence of overflow. Rounding errors come from the fact that each multiplication does not give the full 64-bit product AB but only its highest part $\lfloor AB/2^{32} \rfloor$. For a given evaluation order, this analysis of rounding errors and overflows can be done automatically using the tool Gappa [29, 30, 10].

Because of distributivity and associativity, the number μ_d of all possible evaluation orders of $P(s,t)$ grows extremely fast with the degree d. The first values have been computed recently [31] and, in the above example of square root where $d = 8$, one has

$$\mu_8 = 1055157310305502607244946 \approx 10^{24}$$

different schemes. Among all these schemes, we want one having the lowest latency while satisfying a prescribed rounding error bound.

Since exhaustive search is out of reach, heuristics have been used instead, leading to the design of a tool called

RIGHTSLINK

CGPE (Code Generation for Polynomial Evaluation) [37, 38]. CGPE currently implements some heuristics allowing to quickly produce evaluation schemes (in the form of portable C code) that are accurate enough and have reduced latency (and a reduced number of multiplications) on a target like the ST231. It also computes lower bounds on the latency, which made it possible to conclude in our cases (square root, division) that the retained schemes are optimal (square root) or 1 cycle from the optimal (division). The accuracy/overflow certificates are produced by Gappa.

As an example, we show in Figure 1 and Listing 1 the scheme that CGPE has found in the case of square root and the corresponding C code it has generated. Its latency on

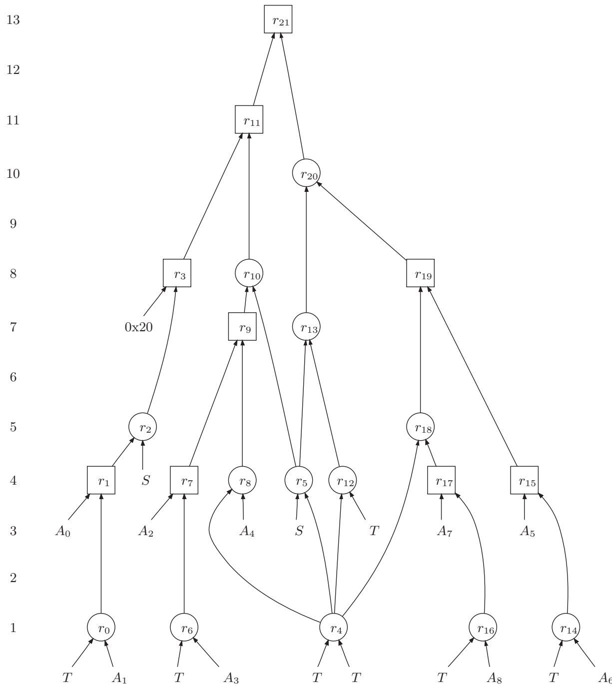

Figure 1: Generated evaluation scheme for square root using the bivariate polynomial approach of [20].

ST231 is of 13 cycles, which matches the computed lower bound. For comparison, Horner's rule, which is fully sequential, takes 36 cycles on ST231. Interestingly, the 13-cycle scheme uses only 4 more multiplications compared to Horner's rule (which is known to be the evaluation order minimizing the number of multiplications [8]).

Finally, it should be noted that the combination of the three tools Sollya, CGPE, and Gappa allows us to tackle more functions than just square root and division, and several other implementations have been [24, 38] or are currently being written using them.

6. COMPILER OPTIMIZATION

As we have seen in Section 2.3, the compiler is key to generate efficient target code. As many techniques are classical or have been described elsewhere (see for example [32]), we will focus on describing the 'Range Analysis' framework, and show a motivating case for its use in a specific optimization linked with the code generation for floating-point support.

Listing 1: Generated C code for the scheme of Fig. 1.

```
uint32_t r0  = mul(T, 0x3ffffafc);
uint32_t r1  = 0x80000007 + r0;
uint32_t r2  = mul(S, r1);
uint32_t r3  = 0x00000020 + r2;
uint32_t r4  = mul(T, T);
uint32_t r5  = mul(S, r4);
uint32_t r6  = mul(T, 0x07f9a6be);
uint32_t r7  = 0x0fff6f59 - r6;
uint32_t r8  = mul(r4, 0x04db72ce);
uint32_t r9  = r7 + r8;
uint32_t r10 = mul(r5, r9);
uint32_t r11 = r3 - r10;
uint32_t r12 = mul(T, r4);
uint32_t r13 = mul(r12, r5);
uint32_t r14 = mul(T, 0x0198e4c7);
uint32_t r15 = 0x0304d2f4 - r14;
uint32_t r16 = mul(T, 0x0019b4c0);
uint32_t r17 = 0x0093fa25 - r16;
uint32_t r18 = mul(r4, r17);
uint32_t r19 = r15 + r18;
uint32_t r20 = mul(r13, r19);
uint32_t r21 = r11 + r20;
```

Range analysis is a variant of the analysis used by constant propagation algorithm [39] operating on SSA where we bound the value (possibly) taken by any variable with a value (range) in a lattice.

We will briefly give an overview of this constant propagation algorithm.

For constant propagation the lattice contains, in addition to initial constant values, the following specific values:

- $\top$ meaning that the variable is unvisited,
- $\bot$ meaning that the value is unknown.

The algorithm proceeds by visiting instructions and lowering the lattice values on the variables according to the following meet ($\sqcap$) rules, as more information is discovered, until a fixed-point is reached:

$$\begin{aligned}&\text{any} \sqcap \top = \text{any}\\&\text{any} \sqcap \bot = \bot\\&c_1 \sqcap c_1 = c_1\\&c_1 \sqcap c_2 = \bot \text{ if } c_1 \neq c_2\end{aligned}$$

In its simplest form, the range analysis uses an extension of the constant propagation lattice: the lattice represents the ranges of possible values of a variable, the meet rules are an extension of constant propagation, and the constant propagation algorithm can be used almost unchanged.

With $[x, y]$ used here to represent the range of integer values between x and y, the following rule replaces the constant rule:

$$[x_1, y_1] \sqcap [x_2, y_2] = [\min(x_1, x_2), \max(y_1, y_2)]$$

A whole family of range analyzes can be defined with slightly different range lattices: for instance analyzes operating on used bit values. In all cases the framework remains the same, only the lattice implementation changes.

It is also useful to have a backward analysis, using the same lattices as for the forward analysis but visiting the instructions backward. This enables for instance the computation of ranges of values needed by the use of a variable, useful to remove useless sign extensions.

RIGHTSLINK

In the Open64 compiler, these analyzes are split into generic and target specific parts. First target specific parts are used to handle unusual, target specific instructions (for instance the ST200 clz instruction creates values in the $[0, 32]$ range regardless of its input). Then target independent part acts on generic instruction types based on standard Open64 compiler predicates.

After the range analysis has assigned a value range to each variable, this information is used by the Range Propagation phase to perform various code improvements, that are first target specific, then generic.

For the ST200, the target specific parts include:

- re-selection of specific instructions, such as lower precision multiplication;
- generalized constant folding (cases leading to constant results whereas operands are not constant);
- long to short immediate transformations;
- shift-or transformations using the shift-add ST200 instruction;
- non-wrapping subtract to zero;
- constant result cases for special multiplications.

The generic parts generally tend to create dead code that is further removed by later phases:

- constant propagation (if the variable is found in the range $[x, x]$);
- removal of unnecessary sign/zero extensions;
- removal of unnecessary min/max instructions.

Range propagation is indeed very similar to a 'peepholing' transformation, where the knowledge of ranges on operands of operations enables more powerful and precise transformations.

One interesting example that we implemented in the compiler is the following, an oversimplified piece of code that we generated for emulating the single precision floating-point division:

```
 1 inline
 2 uint32_t minu(uint32_t a, uint32_t b)
 3                 { return a < b ? a : b ; }
 4
 5 uint32_t test5r(uint32_t x, uint32_t y,
 6                              int32_t z) {
 7   int32_t C,u;
 8   if(z>3) u = 6; else u = 1;
 9   C = minu(y,2);
10   return x >> (u+C);
11  }
```

On the ST200, the right shift expression line 10, $x >> (u + C)$ (or similarly with a left shift), can be transformed into $(x >> C) >> u$, which improves the parallelism by relaxing the data dependency on u, provided that the following conditions hold:

$$u \in [0, 31], \quad C \in [0, 31], \quad u + C \in [0, 31].$$

This transformation is done in the target specific part of the range analysis, since obviously we can enable it only when can prove the preconditions.

Then, instead of $x >> (u + C)$ that incurs the following computations:

[1] computation of u
[2] $\mathrm{tmp} = u + C$
[3] $x >> \mathrm{tmp}$

we get a potentially better use of ILP (|| here means "in parallel with"):

[1] $\mathrm{tmp} = x >> C$ || computation of u
[2] $\mathrm{tmp} >> u$

For instance, the above `test5r` function now takes 3 cycles instead of 4 with this optimization.

7. CONCLUDING REMARKS

Let us conclude with remarks on the validation of the numerical quality of the codes produced by the techniques and tools presented so far. Validating a floating-point implementation that claims to be IEEE 754 compliant is often tricky. For the *binary32* format, for which every data can take 2^{32} different values, exhaustive testing is limited to univariate functions. For example, the square root code of FLIP can be compiled with Gcc under Linux and compared exhaustively against the square root functions of GNU C (glibc[3]) or GNU MPFR[4] within a few minutes.

However, this is not possible anymore for bivariate functions like $+$, $-$, $\times$, $/$. To get higher confidence, a first way is to use some existing test programs for IEEE floating-point arithmetic like the *TestFloat* package [16] and, for division in particular, the *Extremal Rounding Tests Set* [28].

A second, complementary way is to get higher confidence, already at the design stage, in the algorithms used for each subtask of the high-level descriptions of Section 3. The techniques and tools that have been reviewed make this possible as follows:

- for the most regular parts of the computation (*i.e.*, parallel polynomial evaluation schemes), we rely on the automatic error analysis functionalities offered by tools like Sollya and Gappa;
- for other subtasks (like special-value filtering and handling, rounding algorithms, computation of the sign and exponent of the result), we rely on proof-and-paper analysis written in terms of the parameters p, k, ... of the format. Typical examples are the symbolic expressions in (5), (6), and (7).

Our experience with the implementation of floating-point arithmetic shows that this kind of symbolic analysis can be really helpful to produce algorithms and codes that are not only faster but also *a priori* safer. Furthermore, establishing properties parameterized by the format should allow to scale easily from, say, binary32 to binary64 implementations. A future direction in this area could be to automate the derivation of such symbolic properties. Another direction is about automatic numerical error analysis: although CGPE produces C codes for polynomial evaluation that have a guaranteed accuracy, we have no guarantee that compilation will preserve the order of evaluation, thus potentially spoiling the accuracy and so the correctness of the whole implementation.

[3]`http://www.gnu.org/software/libc/`
[4]`http://www.mpfr.org/mpfr-current/`

RIGHTSLINK
Copyright Clearance Center

Therefore, another direction is to explore the possibility of certifying numerical accuracy not only at the C level but also at the assembly level.

8. ACKNOWLEDGMENTS

This research was supported by "Pôle de compétitivité mondial" *Minalogic* (Sceptre Project) and by the ANR project *EVA-Flo*. This research effort is going on with an enlarged scope in the *Mediacom* project, which is part of the French government NANO-2012 R&D funding program.

9. REFERENCES

[1] IEEE standard for floating-point arithmetic. IEEE Std. 754-1985, 1985.

[2] IEEE standard for floating-point arithmetic. IEEE Std. 754-2008, pp.1-58, August 2008.

[3] C. Artigues, S. Demassey, and E. Neron. *Resource-Constrained Project Scheduling: Models, Algorithms, Extensions and Applications*. ISTE, 2008.

[4] C. Bertin, N. Brisebarre, B. D. de Dinechin, C.-P. Jeannerod, C. Monat, J.-M. Muller, S. K. Raina, and A. Tisserand. A Floating-point Library for Integer Processors. In *Proceedings of SPIE 49th Annual Meeting International Symposium on Optical Science and Technology, Denver*, volume 5559, pages 101–111. SPIE, Aug. 2004.

[5] B. Boissinot, A. Darte, B. Dupont de Dinechin, C. Guillon, and F. Rastello. Revisiting out-of-SSA translation for correctness, code quality, and efficiency. In *International Symposium on Code Generation and Optimization (CGO'09)*, pages 114–125. IEEE Computer Society Press, Mar. 2009.

[6] R. Brent and P. Zimmermann. *Modern Computer Arithmetic*. Mar. 2010. Version 0.5.1.

[7] C. Bruel. If-conversion for embedded VLIW architectures. *International Journal of Embedded Systems*, 4(1):2–16, 2009.

[8] B. Bürgisser, C. Clausen, and M. Shokrollahi. *Algebraic Complexity Theory*, volume 315 of *Grundlehren der mathematischen Wissenschaften*. Springer-Verlag, 1997.

[9] S. Chevillard and C. Lauter. Sollya. Available at `http://sollya.gforge.inria.fr/`.

[10] M. Daumas and G. Melquiond. Certification of bounds on expressions involving rounded operators. *Transactions on Mathematical Software*, 37(1), 2009.

[11] B. Dupont de Dinechin. Time-Indexed Formulations and a Large Neighborhood Search for the Resource-Constrained Modulo Scheduling Problem. In P. Baptiste, G. Kendall, A. Munier-Kordon, and F. Sourd, editors, *3rd Multidisciplinary International Scheduling conference: Theory and Applications (MISTA)*, 2007. http://www.cri.ensmp.fr/classement/2007.html.

[12] M. D. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kaufmann, 2004.

[13] J. A. Fisher, P. Faraboschi, and C. Young. *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann, 2005.

[14] FLIP (Floating-point Library for Integer Processors). Available at `http://flip.gforge.inria.fr/`.

[15] C. Guillon, F. Rastello, T. Bidault, and F. Bouchez. Procedure placement using temporal-ordering information: dealing with code size expansion. *Journal of Embedded Computing*, 1(4):437–459, 2005.

[16] J. Hauser. The SoftFloat and TestFloat Packages. Available at `http://www.jhauser.us/arithmetic/`.

[17] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.

[18] 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 the 14th IEEE Symposium on Computer Arithmetic*, pages 233–240, Adelaide, Australia, 1999.

[19] C.-P. Jeannerod, H. Knochel, C. Monat, and G. Revy. Faster floating-point square root for integer processors. In *IEEE Symposium on Industrial Embedded Systems (SIES'07)*, Lisbon, Portugal, July 2007.

[20] C.-P. Jeannerod, H. Knochel, C. Monat, and G. Revy. Computing floating-point square roots via bivariate polynomial evaluation. Technical Report RR2008-38, Laboratoire de l'Informatique du Parallélisme (LIP), 46, allée d'Italie, F-69364 Lyon cedex 07, France, October 2008.

[21] C.-P. Jeannerod, H. Knochel, C. Monat, G. Revy, and G. Villard. A new binary floating-point division algorithm and its software implementation on the ST231 processor. In *Proceedings of the 19th IEEE Symposium on Computer Arithmetic (ARITH'19)*, Portland, Oregon, USA, June 2009.

[22] C.-P. Jeannerod, N. Louvet, J.-M. Muller, and A. Panhaleux. Midpoints and exact points of some algebraic functions in floating-point arithmetic. Technical Report RR2009-26, LIP, Aug. 2009.

[23] C.-P. Jeannerod, S. K. Raina, and A. Tisserand. High-radix floating-point division algorithms for embedded VLIW integer processors. In *17th World Congress on Scientific Computation, Applied Mathematics and Simulation IMACS*, Paris, France, July 2005.

[24] C.-P. Jeannerod and G. Revy. Optimizing correctly-rounded reciprocal square roots for embedded VLIW cores. In *Asilomar'09: Proceedings of the 43rd Asilomar Conference on Signals, Systems, and Computers*, Washington, DC, USA, 2009. IEEE Computer Society.

[25] W. Kahan. IEEE 754: An interview with William Kahan. *Computer*, 31(3):114–115, Mar. 1998.

[26] W. Kahan. A brief tutorial on gradual underflow. Available as a PDF file at `http://www.cs.berkeley.edu/~wkahan/ARITH_17U.pdf`, July 2005.

[27] P. Markstein. *IA-64 and Elementary Functions: Speed and Precision*. Hewlett-Packard Professional Books. Prentice-Hall, Englewood Cliffs, NJ, 2000.

[28] D. W. Matula and L. D. McFearin. Extremal rounding test sets. Available at `http://engr.smu.edu/~matula/extremal.html`.

[29] G. Melquiond. Gappa - génération automatique de preuves de propriétés arithmétiques. Available at

RIGHTSLINK

http://lipforge.ens-lyon.fr/www/gappa/.

[30] G. Melquiond. *De l'arithmétique d'intervalles à la certification de programmes*. PhD thesis, École normale supérieure de Lyon, France, November 2006.

[31] C. Mouilleron. Sequences A173157 and A169608. The On-line Encyclopedia of Integer Sequences (OEIS), February 2010. Available at http://www.research.att.com/~njas/sequences/A173157.

[32] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[33] J.-M. Muller. *Elementary functions: algorithms and implementation*. Birkhäuser, second edition, 2006.

[34] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.

[35] M. L. Overton. *Numerical computing with IEEE floating point arithmetic*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2001.

[36] S.-K. Raina. *FLIP: a Floating-point Library for Integer Processors*. PhD thesis, École normale supérieure de Lyon, France, September 2006.

[37] G. Revy. CGPE - Code Generation for Polynomial Evaluation. Available at http://cgpe.gforge.inria.fr/.

[38] G. Revy. *Implementation of binary floating-point arithmetic on embedded integer processors: polynomial evaluation-based algorithms and certified code generation*. PhD thesis, Université de Lyon - École Normale Supérieure de Lyon, France, December 2009.

[39] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, 1991.

RIGHTSLINK