厳密な浮動小数点演算セマンティクスのJava実行時コンパイラへの実装

首藤一幸[†] 関口智嗣[†] 村岡洋一[††]

IA-32プロセッサは，IEEE 754準拠であるにもかかわらず，ある浮動小数点演算に対して他のプロセッサとは異なる結果を返す．IA-32プロセッサ上で他のプロセッサと同一の演算結果を得るための対処をJava Just-in-Timeコンパイラに実装した．倍精度数の演算ではストア–リロードとスケーリングを行う必要があるが，単精度数の演算では丸め精度を倍精度としたままストア–リロードだけ行えば十分であることが明らかになった．また，いくつかの実装方法について性能への影響を調べたところ，スケーリング専用命令ではなく乗算命令を用いることで性能の低下幅は約40%にまで抑えられることが分かった．

Efficient Implementation of Strict Floating-point Semantics

Kazuyuki Shudo,[†] Satoshi Sekiguchi[†] and Yoichi Muraoka[††]

IA-32 processors yield different results of floating-point operations from other processors, even though they are all compliant with IEEE 754. The Java specifications need runtime systems to implement the FP-strict semantics, which other IEEE 754 compliant processors naturally fulfill. We implemented the semantics on a Java Just-in-Time compiler for IA-32. This study reveals that single-precision operations can be performed with precision control bits of the processor staying as double-precision. Performance evaluation demonstrates that our implementation method reduced the performance decline by the semantics down to 40%.

1. はじめに

最近のほとんどのプロセッサはIEEE 754[1]に準拠している．これは浮動小数点演算の規格で，浮動小数点数の表現形式や，四則演算，平方根などの基本的な操作，また，丸め，例外の規定を含んでいる．事実，SPARC，Alpha，PowerPC，MIPS，IA-32（通称x86）はすべてIEEE 754準拠である．それにもかかわらず，IA-32は，ある演算に対して上記の他のアーキテクチャとは異なる結果を返す．IA-32のこの特殊な仕様によって，演算を行う計算機によって結果が異なるという，計算の再現性の問題が起こりうる．また，異機種混在環境での並列，分散計算では，サブタスクの各プロセッサへの割当てがどうなされるかによって結果が異なるという問題が起こりうる．

処理結果の再現性を重視しているJava言語，Java仮想マシン[2]にとってもIA-32の仕様は問題であった．異機種間での再現性を保証するため，Java言語仕様[3]はSPARCなどと同じセマンティクスを浮動小数点演算に要求していた．IA-32でこのセマンティクスを達成するには演算に加えて補正処理が必要であり，IA-32用のJava処理系は性能上のペナルティを強いられていた．とはいえ，現実には各処理系ベンダはこの仕様を実装せず，Intel社，IBM社ほかの要求によって仕様の側が緩和されるに至った．

Java 2より，言語仕様に`strictfp`という，クラス，メソッド，インタフェース修飾子が導入された[4]．`strictfp`のレキシカルスコープ内では従来どおりのセマンティクスが要求される反面，スコープ外，つまり`strictfp`が指定されない限りは，IA-32そのままのセマンティクスも許容されるようになった．それでも，`strictfp`のスコープ内では補正が必要である．

`strictfp`のためのこの補正処理にはいくつかの実装方法がある．我々はそれらをJavaバイトコードの実行時（JIT）コンパイラ[5),6)]に実装し，それぞれについて性能への影響を調べた．また，特定のJITコンパイラにおける影響だけでなく実装法それ自身のペナ

† 産業技術総合研究所
National Institute of Advanced Industrial Science and Technology
†† 早稲田大学理工学部
School of Science & Engineering, Waseda University

ルティを調べる目的で，Java 言語で記述したものと同等のベンチマークプログラムをアセンブリコードでも記述した．

本論文では，まず，問題となる IA-32 独特の仕様を説明し，そのうえで，Java 言語の `strictfp` が規定する他のプロセッサと同じセマンティクスを達成する手法を述べる．続いて，いくつかの実装方法を提案し，それらの効率を比較する目的で行った性能評価の結果を示す．また，7 章では，Java 言語仕様の改訂後も残された問題を指摘する．

2. IA-32 の特徴的な仕様

IEEE 754 では，正規化数（normalized number）は次のように表現される．

$$(-1)^s 2^E (1.b_1 b_2 \cdots b_{p-1})$$

ここで，仮数部は 2 進数であり，他は 10 進数である．それぞれの記号の意味は次のとおりである．

- s : 符号ビット（0 か 1）
- b_i : 仮数部中の 1 ビット（0 か 1）
- p : 仮数部の精度［bit］
- E : 指数部

IA-32 は浮動小数点数の表現形式として，IEEE 754 が定める単精度（32 ビット），倍精度（64 ビット）のほかに，独自の拡張倍精度（80 ビット）を持つ（**表 1**，**図 1**）．IEEE 754 はこの拡張形式を許している．

IA-32 プロセッサで x87 浮動小数点演算命令を用いる場合，プロセッサ内ではつねに，浮動小数点数は 80 ビットの拡張倍精度で保持される．また，他の多くのアーキテクチャは精度に応じた演算命令を持っているところ，x87 命令には精度ごとの演算命令がない．たとえば乗算であれば，SPARC なら `fmuls` 命令と `fmuld` 命令，MIPS なら `mul.s` 命令と `mul.d` 命令を持ち，演算結果は命令に応じた精度に丸められる．丸めとは，結果が表 1，図 1 の形式で表現できないものとなった場合，表現可能な形式（たとえば最も近い値）を選ぶことである．x87 命令では，その代わりに丸め精度を単精度，倍精度，拡張倍精度のいずれかに設定可能であり，この設定に従って演算結果が丸められる．ここで IA-32 を特徴付けているのは，丸め精度が仮数部にしか影響を与えないという仕様である．丸め精度が何であれ，指数部はつねに拡張倍精度と同じ 15 ビットで表現される（図 1）．他のアーキテクチャでは，単精度数の演算であれば 8 ビットに，倍精度であれば 11 ビットに丸められる．ところが IA-32 内部では指数部はつねに 15 ビットなので，他のアーキテクチャでは起こる溢れが発生しない場合がある．たとえば，レジスタ *reg* に対する次の操作を順に行った場合，他のアーキテクチャでは途中でオーバフローが起こりレジスタの値は $+\infty$ となるが，IA-32 で x87 命令を用いるとオーバフローは起こらず，2^{+1023} が得られる．

（1） *reg* に 2^{+1023} をロード
（2） *reg* に 2^{+1023} を加算
（3） *reg* から 2^{+1023} を減算

`strictfp` の文脈では，IA-32 でも他のアーキテクチャと同じ挙動が要求されるので，何らかの工夫が必要となる．

ここで述べた IA-32 の仕様は，Intel 社以外の互換プロセッサ，たとえば AMD 社の K6 や Athlon，Transmeta 社の Crusoe にも共通のものである．

表 1 各精度の諸元
Table 1 The properties of each precision of IA-32.

	単精度	倍精度	拡張倍精度
全体の長さ［bit］	32	64	80
仮数部の精度（p）† ［bit］	24	53	64
指数部（E）の精度［bit］	8	11	15
E の最大値	+127	+1023	+16383
E の最小値	−126	−1022	−16382

† 単精度と倍精度はそれぞれ 23，52 ビットに加えて隠れた最上位ビット（つねに 1）を持つので，精度は 24，53 ビットだが，IA-32 の拡張倍精度にはこの隠れた 1 がないため，精度はビット数と同じ 64 ビットである．

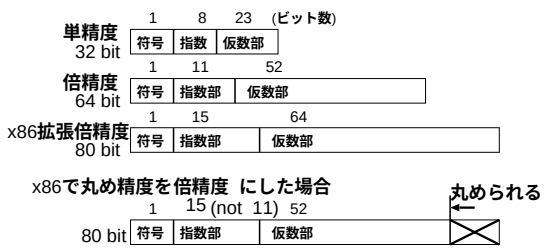
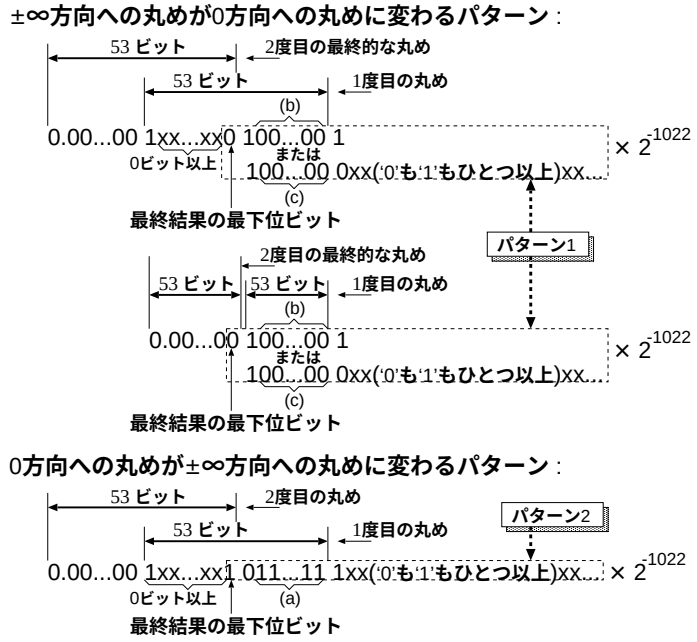


図 1 浮動小数点数の表現形式と IA-32 の丸め精度
Fig. 1 Floating-point value formats of IA-32.

3. Golliver の手法

IA-32 上で，`strictfp` の要求する他のアーキテクチャと同じセマンティクス（*FP-strict* [4] と呼ぶ）を達成する手法が Golliver によって提案されている[7]．これは，現在知られている最も効率の良い手法で，Java Grande の Numerics Working Group も推奨している．本論文は，この手法の様々な実装方法を提案し，性能評価の結果を報告するものである．

FP-strict な演算を達成するための最も素朴な方法は，プロセッサの整数命令を使って浮動小数点演算を模することであろう．こういったソフトウェアライブラリは数多く存在する．しかしこの方法では性能の低



(注: ビットパターンはすべて2進表現. '0...0' は '0' の連続. 'x' は '0' か '1'.)

図 2 2 度丸めの影響を被る倍精度数
Fig. 2 Binary expression of double-precision values that suffers double rounding.

さが問題となる．浮動小数点演算用ハードウェアでの処理と比較して，数倍どころでは済まず，10 倍，100 倍といったオーダで性能が低下する．それに対して，Golliver 手法のペナルティはたかだか 2 倍から 4 倍と予測されている[7]．

Golliver の手法は，ストア–リロード（*store-reload*）とスケーリング（*scaling*）という 2 つの部分からなる．

3.1 ストア–リロード

FP-strict に沿った結果を得るために，まず，指数部の溢れを適切に起こすことを考える．つまり，レジスタ上ではつねに 15 ビットである指数部を，倍精度なら 11 ビット，単精度なら 8 ビットで溢れさせる．これは，目的に応じた精度としてレジスタからメモリへストアし，それを再びレジスタにロードすることで達成できる．メモリ上でいったん，拡張倍精度ではなく単精度や倍精度として表現させることで，指数部を強制的に丸めるのである．この操作は，加減乗除のすべてで必要である．

3.2 2 度丸めという問題

しかしストア–リロードだけでは完全ではない，演算時とメモリへのストア時の 2 回，値が丸められて，仮数部が FP-strict な結果とは異なってしまうことがある．すなわち，アンダフローが起きて，倍精度としては非正規化数（denormalized number）として表されるべき値が，指数部が 15 ビットと余分にあるためにレジスタ上では正規化数（normalized number）として表されてしまい，メモリへのストア時になって初めて非正規化数に変換される場合である．

非正規化数とは，通常の表現形式である正規化数（2 章）のうち絶対値が最小である値と 0 の間の値を表現するための表現形式であり，IEEE 754 に含まれる．次のように表現される 0 以外の値が非正規化数である．ここで，記号の意味は 2 章と同じである．

$$(-1)^s 2^{\mathrm{E}\text{ の最小値}}(0.b_1 b_2 \cdots b_{p-1})$$

浮動小数点演算の結果はアンダフローすると +0 や −0 と表現される．しかし，本来は 0 ではない値が桁落ちなどによるアンダフローで 0 と表現されてしまうことは，その結果を用いる続く演算に大きな誤差をもたらしかねない．そこで，正規化数の範囲を超えた場合でも漸進的に（gradual）アンダフローさせるための緩衝材として，IEEE 754 では非正規化数を規定している．

この 2 度丸めが起きると，演算時に 1 度で非正規化数にまで丸められた場合とは結果が異なってしまうことがある．1 度目の丸めによって 2 度目の最終的な丸めの方向が変わるのである．単精度演算については 4.2 節で考察するので，ここでは倍精度演算について

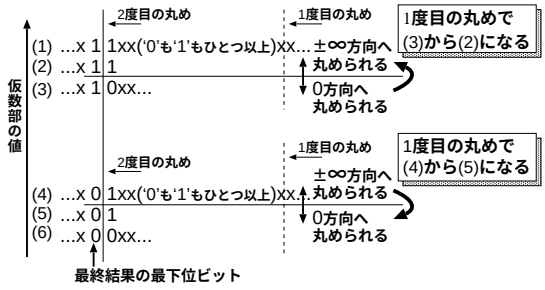


図 3 2 度丸めが問題となる理由
Fig. 3 The mechanism of double rounding problem.

考える．

図 2 に，この 2 度丸めによって，FP-strict ではなくなってしまう演算結果の形式を示す．図中の（a），（b），（c）については，図 4 と対応付けて後述する．この形式の値は倍精度では最終的に非正規化数として表されるが，指数部が 15 ビットある IA-32 のレジスタ上では正規化数として表現できてしまう．つまり，演算の直後，レジスタ上では正規化数となっている．その後，メモリへストアする際に非正規化数に変換され，ここで 2 度目の丸めが起きる．なお，加減算では 2 度丸めの問題は起きない．倍精度数どうしの加減算の丸め前の結果は図 2 の形式をとりえないからである．

以下，図 2 の形式をいかに導くかを示す．本論文では，プロセッサの丸めモードは IEEE 754 の既定値である最近値への丸めのままになっていると仮定する．2 度丸めが問題となるのは，1 度目の丸めによって 2 度目の丸めの方向が変わってしまう場合である．そのため，もし，1 度目の丸めが 2 度目の丸めの方向を変えないならば，最終的に得られる値は 1 度で非正規化数にまで丸めた場合と同じであり，2 度丸めは問題とはならない．丸めの方向が変わるとは，0 方向に丸められる値が $+\infty$ または $-\infty$ 方向に丸められる値に変わるか，その逆が起こることを指す．

図 3 に示す 6 種類のビットパターンは，仮数部がとりうる形式を網羅している．（2）と（5）は結果の最下位ビットより下位のビット列が同じ（1 のみ）ではあるが，（2）は ∞ 方向へ，（5）は 0 方向へ丸められることに注意していただきたい．これは，最近値への丸めでは，2 つの最も近い表現可能な値が等しく近しいときは，最下位ビットが 0 であるものが選択される，という IEEE 754 の規定による．ここで興味があるのは，1 度目の丸めによって 2 度目の丸めの方向が変わるかどうか，つまり，（1），（2），（4）の形式が，1 度目の丸めによって（3），（5），（6）の形式に変わるか，またはその逆に変わることがあるかどうか，である．このうち実際に起こりうるのは，（3）から（2）への丸めと，（4）から（5）への丸めだけである．（1）から（6）のうちで 2 度丸めの影響を被る演算結果の形式は，

- 1 度目の丸めで（2）になるような（3）の形式
- 1 度目の丸めで（5）になるような（4）の形式

に限られる．

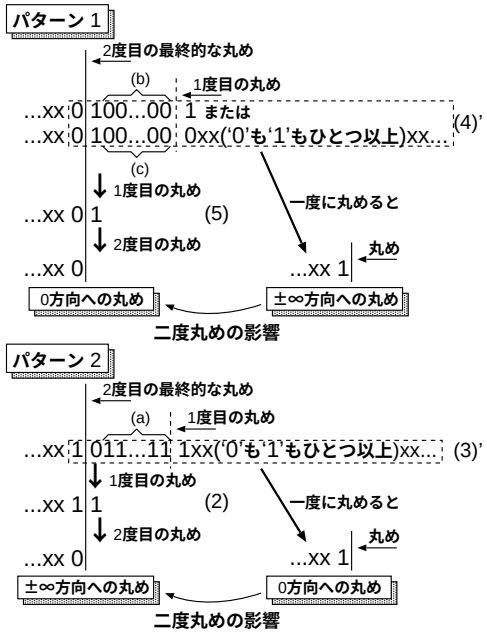


図 4 2 度丸めの影響を被る仮数部の形式
Fig. 4 The bit patterns of significand that suffers double rounding problem.

図 4 は，このように，（3），（4）の形式でありながら，1 度目の丸めで（2），（5）の値に丸められるという条件を満たす仮数部の形式を示したものである．（3）’，（4）’ はそれぞれ，（3），（4）に，さらに，必要なすべての制約を加えた形式である．（a）が 1 の連続，（b）と（c）が 0 の連続であるのは，1 度目の丸めでそれぞれ（2），（5）にまで丸められるために必要な条件である．丸め前の値と丸められた結果の差は，結果の最下位ビット 1 ビット分より小さいので，（3）’ と（2），（4）’ と（5）の差をそれより小さくするためには，（a）は 1 の連続，（b）と（c）は 0 の連続でなければならないのである．また，（3）’，（4）’ 中で，（a），（b），（c）より下位のビットパターンに課せられている条件は，それぞれ ∞ 方向，0 方向への丸めとなるための条件である．

最後に，図 4 のビットパターンにおいて，0 や 1 が連続する部分（a），（b），（c）のビット長について考察する．（a）と（c）は，長さが 0 ビットであっても 1 度目の丸めでそれぞれ（2），（5）の形式となる．しかし，（b）が 0 ビットだった場合，丸めた結果は（5）にはならない．この場合，2 度に分けて丸められても 1 度に丸められても同じ結果となる．2 度丸めの影響を被らないのである．つまり，（a）と（c）は 0 ビット以上だが，（b）だけは，1 ビット以上の場合のみ 2 度丸めが問題となる．

こうして構成した図 4 のビットパターンに，倍精度演算ゆえのビット幅の制約を加えたものが図 2 となる．図 4 の（4)' と（3)' は，それぞれ図 2 のパターン 1 とパターン 2 に対応する．図 2 でも図 4 と同じく，（a）と（c）は 0 ビット以上であり，（b）は 1 ビット以上である．

次に，2 度丸めの問題によって結果が変わってしまう演算の具体例をあげる．( ) 内は IEEE 754 倍精度形式（図 1）で表した場合のビット列（16 進表記）である．

乗算：

$1.112808544714844 \times 10^{-308}$
$\times 1.0000000000000002$
(0x0008008000000000
$\times$ 0x3FF0000000000001)

除算：

$2.225073858507201 \times 10^{-308}$
$\div 0.9999999999999999$
(0x000FFFFFFFFFFFF
$\div$ 0x3FEFFFFFFFFFFFFF)

3.3 スケーリング

この，ストア–リロードにともなう 2 度丸めを防ぐためには，演算の時点で適切にアンダフローが起き，非正規化数が得られればよい．指数部が 15 ビットではなく 11 ビットであるかのようにアンダフローを起こすために，オペランドと演算結果に対してスケーリングを行う．一方のオペランドにある定数を乗じておき，演算結果にその定数の逆数を乗じる．これによって，アンダフローが起きる境界を調節できる．この定数は $2^{-16382-(-1022)}$ である．この手法は，非正規化数による漸進的な（gradual）アンダフローという IEEE 754 の規定も満足させる．

4. JIT コンパイラへの実装

3 章で説明した Golliver の手法を，Java バイトコードの実行時（JIT）コンパイラである shuJIT [5),6)] に実装した．JIT コンパイラが生成するネイティブコードは次の処理を行う必要がある．このためにどのような実装を行ったのかを述べていく．

- 丸め精度を設定する．
- 指数部の溢れを適切に起こす（漸進的アンダフローを含む）．

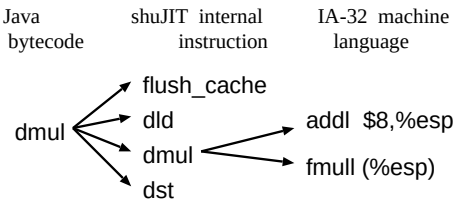


図 5 shuJIT のコード生成手法
Fig. 5 Code generation method of shuJIT.

4.1 JIT コンパイラの概要と変更方法

shuJIT は IA-32 用の JIT コンパイラであり，OS として Linux と FreeBSD，NetBSD をサポートする．**図 5** のように，コンパイル処理の最初に，Java バイトコードの命令列を内部命令に変換する．それに対していくつかの最適化を施し，最後に内部命令をネイティブコード列に置換することでコード生成を行う．ただし，1 つの内部命令に対するネイティブコード列を最大 5 通り用意することで，複数のレジスタを活用できるようにしている[5)]．コード長に対する計算量のオーダが高い処理を避けているために，コンパイル処理が軽いこと，つまり，コンパイル時間に対する性能比が良いことが特徴の 1 つである．

今回，Golliver の手法を実装するためにいくつかの内部命令を追加した．追加したのは，スケーリングを行う命令と，スケーリングに使う定数をレジスタへプリロード（4.3 節）する命令，レジスタから除去する命令である．FP-strict のために必要な場合にこれらの内部命令を生成するようにした．たとえば，乗除算の前後にはスケーリング命令を，プリロードを行う際はメソッドの先頭と末尾にプリロードおよび除去命令を生成する．

また，コンパイラの挙動を変えるオプションを用意した．1 つは `frcstrictfp` で，これを指定すると，あらゆる浮動小数点演算を FP-strict だと見なしてコンパイルする．つまり，つねに `strictfp` 修飾子が指定されているものと見なす．もう 1 つは逆の働きをする `ignstrictfp` で，これが指定された場合 `strictfp` 修飾子を無視する．`frcstrictfp` オプションは性能評価を行う際に有用であった．これを指定することで既存のアプリケーションを変更なしに FP-strict であると

して実行できるためである．

4.2 丸め精度の設定

IA-32 は精度に応じた x87 演算命令を持たず，あらゆる演算に影響を与える丸め精度を設定できるのみである（2 章）．FP-strict な演算を行うためには，この丸め精度の設定にも気を配らなければならない．

最も素朴なのは，演算の前にその演算に応じた精度に設定するという方法だろう．しかし，丸め精度の設定にはメモリアクセスが必要でありそれなりのオーバヘッドがともなうため，設定の回数は極力減らしたい．実は，もし，つねにストア–リロード（3 章）を行うなら，丸め精度は倍精度に設定しておいたままで問題ない．そのまま単精度演算を行っても，3.2 節で述べた 2 度丸めの問題は起きえないため，スケーリングすら不要である．FP-strict な演算を行うためにはいずれにせよストア–リロードは必要であるため，単精度数の演算では，丸め精度を倍精度に設定したままでストア–リロードを行い，スケーリングを行わないという方法が最善である．

丸め精度を倍精度に設定したまま単精度演算を行った状況を考える．まず，IA-32 では指数部の扱いは丸め精度の設定に影響されない（2 章）ので考慮の必要はない．仮数部は，次のとおり 2 回丸められる．

(1) 演算時に倍精度（53 ビット）に．
(2) ストア–リロードでのメモリへのストア時に単精度（24 ビット）に．

ここでもし，このように 2 段階にわたって丸められた結果が 1 度で単精度まで丸められた結果と異なることがあるならば，丸め精度を倍精度のままとしておくことに問題があるということになる．図 4 をもとにして，この種の 2 度丸めで問題が起きる演算結果の形式を構成したものが，**図 6** である．また，**図 7** はビットパターンのみに着目して図 6 を書き直したものである．図 6，図 7 は，この種の 2 度丸めの影響を受ける丸め前の演算結果を網羅している．3.2 節で述べたとおり 2 度丸めの影響を受ける形式は図 4 ですべてであり，この種の 2 度丸めの影響を受けるのは，図 6 に示したように，1 度で丸めた場合に正規化数になる形式と非正規化数になる形式で網羅できるからである．1 度で丸めた場合に正規化数，非正規化数となる値が，それぞれ 2 度丸めの場合にはオーバフロー，アンダフローすることはあるが，その逆は起こらないので，1 度で丸めた場合にオーバフロー，アンダフローする値はこの種の 2 度丸めの影響は受けない．

しかし，四則演算の結果は図 6，図 7 の形式をとりえないため，丸め精度を倍精度としたまま単精度演算

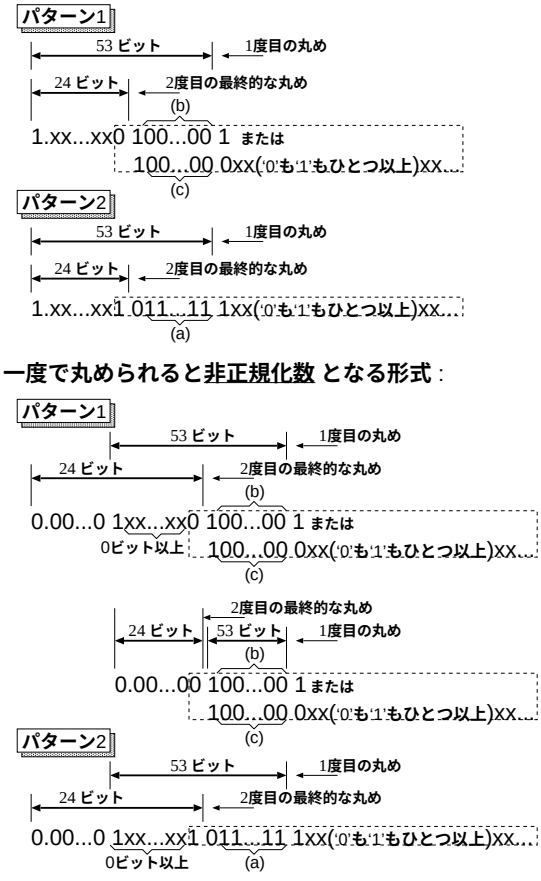


図 6 倍精度，単精度への 2 度丸めで値が変わる仮数部
Fig. 6 The significands that suffers double rounding to double and single precision.

を行ってもまったく問題はない．乗算，加減算，除算と 3 つの場合に分けて，理由を以下に示す．

単精度数の仮数部は最大で 24 ビット幅なので，乗算の真の（丸め前の）結果はたかだか 48 ビット幅である．それに対して，図 6，図 7 の形式は最低でも 54 ビット幅である（最上位と最下位の ‘1’ が 53 ビット離れている）．よって，積は図 6，図 7 の形式をとりえない．

加減算の結果が図 7 のビットパターンをとりうるか否かを調べる．**図 8** は単精度数どうしの加減算について仮数部の処理を筆算で行っている状況を表している．2 度丸めが問題となる形式では，最上位ビットから数えて 54 ビット目以降に ‘1’ がある．加減算の結果で 54 ビット目以降が ‘1’ となるためには，図 8 に示しているように，2 つのオペランドの桁が最低でも 30 ビットは離れている必要がある．このとき，図 8 中の破線で囲んだビット列が，図 7 から導かれる問題のあ

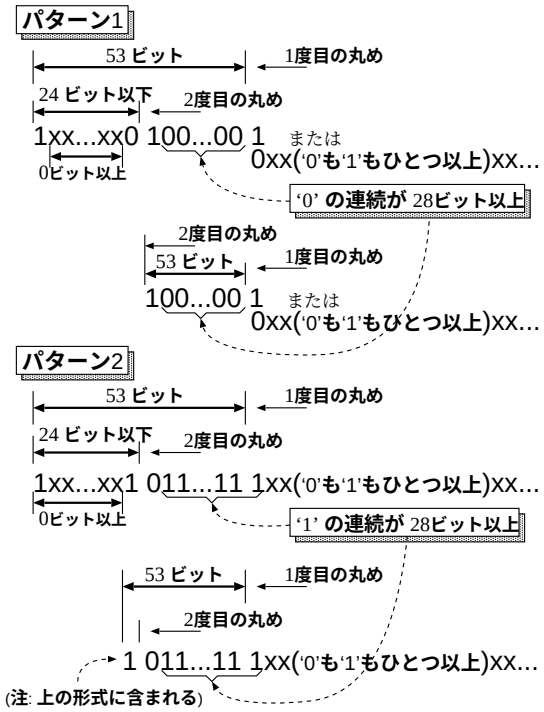


図 7 倍精度，単精度への 2 度丸めで値が変わるビットパターン
Fig. 7 The bit patterns that suffers double rounding to double and single precision.

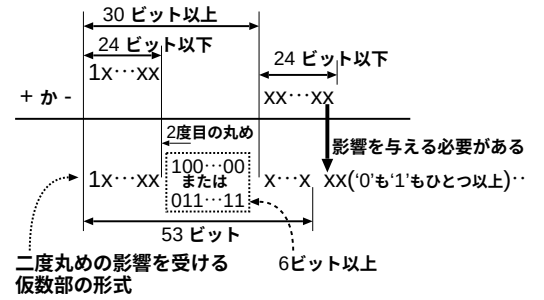


図 8 単精度数どうしの加減算
Fig. 8 Addition and subtraction of single precision numbers.

るビットパターン（1 に続いて 5 ビット以上の 0 か，0 に続いて 5 ビット以上の 1）をとることはありえない．このビット列は，加算ならすべて 0，減算ならすべて 1 となる．ゆえに加減算の結果は図 7 の形式をとりえない．

最後に除算について考える．除算においてこの 2 度丸めが起きないことは，計算機実験を行って確認できた．被除数と除数の仮数部がとりうる値のあらゆる組合せについて，2 度丸めの問題が起きるか否かを調べた結果，起きないことが分かった．実験の詳細は付録 A.1 で述べる．この探索は 1.7 GHz Pentium 4 を用いて 1 カ月程度の処理であり，実際には複数台の計算機を用いて，数日で完了した．

以上により，丸め精度を倍精度としたまま単精度演算を行っても，ストア–リロードにともなう 2 度丸めの問題（3.2 節）は起きないことが分かった．つまり，スケーリングで 2 度丸めを防ぐ必要はなく，ストア–リロードを行うだけで FP-strict となる．

4.3 溢　　れ

JIT コンパイラが生成するコードは，指数部の溢れを適切に起こすために四則演算でストア–リロードを行わなければならない．また，倍精度演算の場合は，ストア–リロードにともなう 2 度丸めの問題を回避するためにスケーリングを行わなければならない（3 章）．

スケーリングを行う方法は，IA-32 の x87 命令の場合，次の 2 通りが考えられる．

- **乗算命令**：2^n を乗じる．
- `fscale` **命令**：2 の巾によるスケーリング命令を用いる．

また，スケーリングに用いる定数をメモリからレジスタにロードするタイミングにもいくつかの選択肢がある．

- スケーリングの直前にロードする．
- プリロードしておく．
  - `strictfp` が指定されたメソッドの先頭．
  - （JIT コンパイラの初期化時）．

JIT コンパイラの初期化時にロードしてしまうと，プログラムの実行中はつねにレジスタが定数に占有されてしまうので，プリロードする際はメソッドの先頭で行うようにした．

スケーリングを乗算で行う場合，定数は拡張倍精度の浮動小数点数でしか表現できない．指数部の絶対値が大きいため，単精度，倍精度数としては表現できないからである．しかし，スケーリングを `fscale` 命令で行う場合は，定数をメモリ上に用意しておく形式として，整数，単精度数，倍精度数の 3 通りが考えられる．333 MHz の Pentium II プロセッサ上で，プリロードを行わずに上記 3 形式を比較した．その結果，単精度数と倍精度数では同じ演算性能が得られたが，それに対して整数では 10^7 回の演算あたり 130 から 140 ミリ秒のオーバヘッドがあった．整数をレジスタ上の浮動小数点数に変換するためのオーバヘッドだと考えられる．この結果に基づき，単精度数を採用した．

今回，2 通りのスケーリング方法と，プリロードの有無を組み合わせた 4 通りの実装を行い，それぞれの性能を比較した．

5. 性能評価

4.3 節で述べたいくつかの実装方法について性能を測定，比較した．比較対象として BulletTrain[8] 2.0.0b15 を用意した．BulletTrain はプログラム実行前にコンパイルを済ませてしまう Ahead-of-Time コンパイラであり，`strictfp` のセマンティクスを正しく実装している数少ない処理系である．

実験環境として，2 種類の IA-32 プロセッサ，1.7 GHz の Pentium 4 と 650 MHz の Mobile Pentium III を用意した．OS は，BulletTrain での実験では Windows 2000 SP2，その他の実験では Linux を用いた．shuJIT とともに用いた Java 処理系は Blackdown Java 2 SE 1.2.2 FCS である．

実験結果の図中に，Pentium 4 上で shuJIT を用いた場合に，*SSE2* という実験条件がある．これは，浮動小数点数の四則演算すべてを x87 命令ではなく SSE2 命令で行った場合の結果である．SSE2 については 6 章で述べる．

5.1 乗除算

まず，ストア–リロードとスケーリングの双方を必要とする倍精度の乗算，除算の性能を測った．10 回乗算または除算を含むコードを 5×10^6 回繰り返して実行して時間を計測した．

C 言語の処理系に FP-strict な演算セマンティクスを実装したとしてどの程度の性能が得られるのかも評価した．とはいえ，C コンパイラに変更を加えたのではなく，Java 言語で記述したものと同じベンチマークプログラムを C 言語でも記述し，それをもとに生成したアセンブリコードに各種スケーリング方法（4.3 節）を実装することで上記の状況を模した．C コンパイラとしては GCC 2.95.2 を用いた．

こういった小規模で作為的なベンチマークプログラムを作成する際は，コンパイラによる最適化，つまり演算の強さの軽減やフロー解析に基づく不要コードの除去などの影響を考慮する必要がある．最適化によって意図したものとは異なる処理となってしまわないように注意を払わなければならない．ここでは，上記の最適化を妨げるために，変数の使用を記述するなどの工夫を施し，コンパイラが生成したコードを見て目的のベンチマーク処理が行われることを確認した．また，乗除算に要する時間は演算のオペランドに依存するので，オペランドが 0 や ∞ になって演算時間が極端に短くなってしまうことも防いだ．

図 9，**10**，**11**，**12** はそれぞれ，Pentium 4 での乗算，Pentium 4 での除算，Pentium III での乗算，Pentium III での除算に要した時間である．4 通りのスケーリング方法のほかに，何も補正処理を行わない場合（raw），ストア–リロードのみを行った場合も測

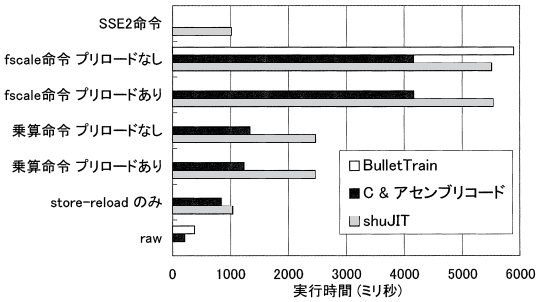


図 9 Pentium 4 での乗算
Fig. 9 Multiplication on Pentium 4 processor.

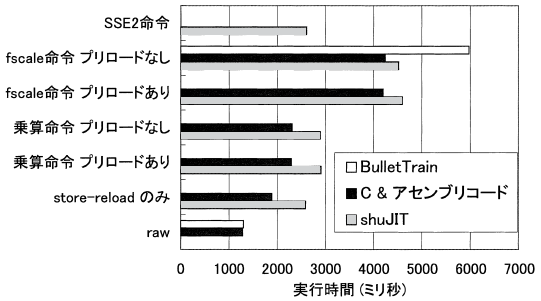


図 10 Pentium 4 での除算
Fig. 10 Division on Pentium 4 processor.

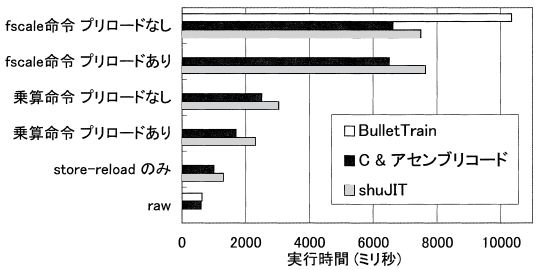


図 11 Pentium III での乗算
Fig. 11 Multiplication on Pentium III processor.

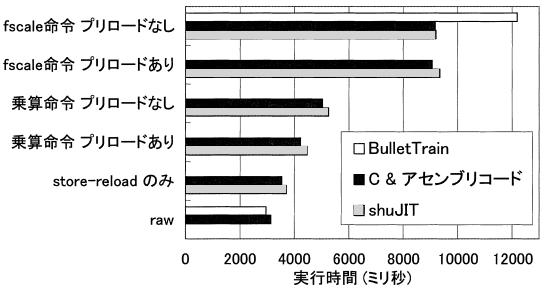


図 12 Pentium III での除算
Fig. 12 Division on Pentium III processor.

表 2 C 言語でのペナルティ
Table 2 Penalty for FP-strictness in C language.

演算	raw		乗算命令，プリロード	ペナルティ
Pentium 4，乗算	209	→	1238	5.92 倍
Pentium 4，除算	1284	→	2299	1.79 倍
Pentium III，乗算	603	→	1694	2.81 倍
Pentium III，除算	3150	→	4233	1.34 倍
	（ミリ秒）		（ミリ秒）	

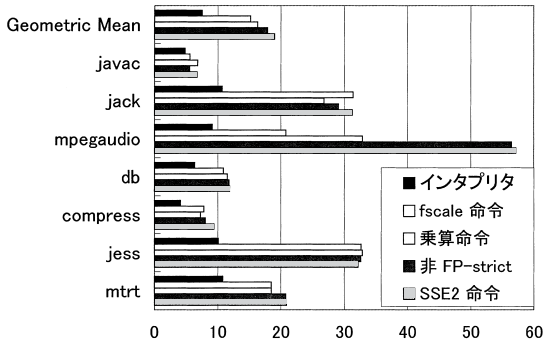


図 13 Pentium 4 での SPEC JVM98
Fig. 13 SPEC JVM98 on Pentium 4 processor.

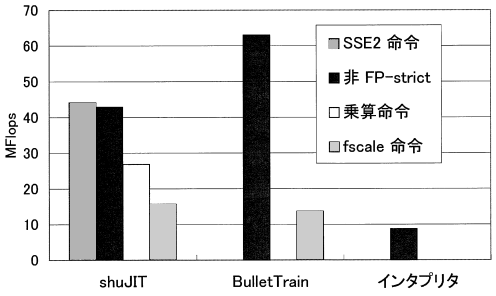


図 14 Pentium 4 での Linpack Benchmark
Fig. 14 Linpack benchmark on Pentium 4 processor.

定した．この 2 者の演算方法は FP-strict ではない．

shuJIT での結果を見ると，スケーリングの方法としては `fscale` 命令を用いるよりも乗算命令を用いた方が，約 2〜3 倍速い．BulletTrain は `fscale` 命令を採用しているのだが，Pentium 4 と Pentium III では乗算命令を用いた方が良いといえる．

表 2 に C 言語での FP-strict のペナルティをまとめる．補正処理なし（raw）の実行時間に対する，補正処理手法の中で最も速かったもの，つまり，乗算命令を用い，プリロードを行った場合の比である．Java Grande Numerics WG は FP-strict によるペナルティは 2 倍から 4 倍と予測しており[7]，表 2 の最大約 6 倍という値はこれを逸脱している．しかし今回の実験方法では補正処理のために追加した命令はコンパイラによる命令スケジューリングの対象となっていないため，実際に C コンパイラに実装した場合は，オーバヘッドはもう少し小さくなると予想される．

5.2 SPEC JVM98 と Linpack Benchmark

図 13，**14**，**15**，**16** はそれぞれ，Pentium 4 での SPEC JVM98，Pentium 4 での Linpack ベンチマーク，Pentium III での SPEC JVM98，Pentium III での Linpack ベンチマークの結果である．いずれも，より大きい値が良い結果を示す．

FP-strict による性能低下がどの程度なのかを調べるため，浮動小数点演算が中心のプログラムについ

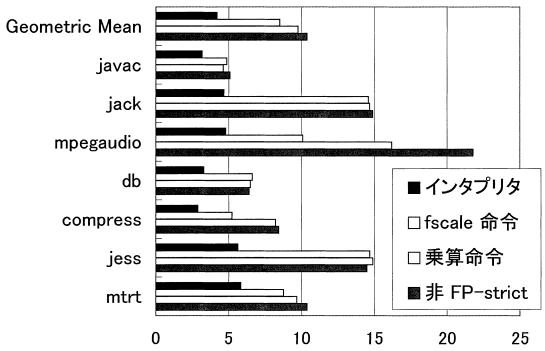


図 15 Pentium III での SPEC JVM98
Fig. 15 SPEC JVM98 on Pentium III processor.

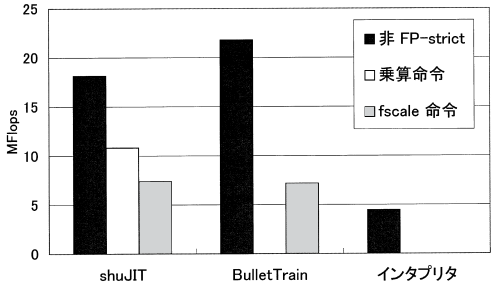


図 16 Pentium III での Linpack Benchmark
Fig. 16 Linpack benchmark on Pentium III processor.

表 3 shuJIT での補正なしに対する FP-strict の性能の比
Table 3 Performance ratio of FP-strict operation to normal operation with shuJIT.

	乗算命令	`fscale` 命令
Pentium 4, mpegaudio	58.2%	36.8%
Pentium 4, Linpack	62.5%	36.6%
Pentium III, mpegaudio	74.3%	46.3%
Pentium III, Linpack	59.6%	40.6%

て，shuJIT での補正処理なしの場合に対する，乗算命令で補正を行った場合の比を算出した．**表 3** に示す．58.2%から 74.3%となっている．悪い場合でもおよそ 60%，つまり 40%の低下である．

一方，`fscale` 命令で補正を行った場合のこれらの値は，36.6%から 46.3%であり，乗算命令を用いた場合と比較して性能が大幅に低下している．5.1 節と同様に，スケーリングには `fscale` 命令よりも乗算を用いた方が良いという結果が得られた．

6. SSE2 命令

IA-32 には，Pentium 4 より Streaming SIMD Extensions 2（SSE2）という命令群が追加された．SSE2 は Single Instruction Multiple Data（SIMD）演算のために用意された命令セットであるが，実は浮動小数点演算全般をサポートしている．表現形式は IEEE 754 の単精度と倍精度であり，四則演算，平方根，丸め方向の制御など，剰余を除いて IEEE 754 の要求をほぼサポートしている[9]．また，複数の値に同一操作を施す SIMD 演算だけではなく Single Instruction Single Data（SISD），つまり 1 つの演算だけを行う命令群も持っている．そのため，これまで IA-32 の浮動小数点演算命令群であった x87 命令の代替として用いることも可能である．ただし，x87 命令が持っている *sin* といった超越関数の命令は持たない．

浮動小数点数の四則演算に x87 命令ではなく SSE2 命令を用いれば，これまで述べてきた Golliver の手法（3 章）を用いるまでもなく，FP-strict に演算を行える．つまり，x87 命令が用いるレジスタは指数部の幅がつねに 15 ビットであるために FP-strict な演算のためには工夫が必要であったが，SSE2 命令が用いるレジスタは 80 ビット拡張倍精度ではなく，また，単精度数と倍精度数では演算命令が分けられているため，特に何も工夫は要らない．

5 章で示した性能評価結果のうち，*SSE2* という結果は，浮動小数点数の四則演算すべてを x87 命令ではなく SSE2 命令で行った場合のものである．いずれの実験においても，x87 命令で FP-strict のための補正処理を何も行わない場合とほとんど同じ性能が得られている．5.1 節の乗算と SPEC JVM98 の jess では SSE2 を使った方が悪い結果となっているが，いずれもたかだか 1%から 3%の悪化である．他のプロセスの影響による誤差があることも考慮すると，x87 命令ではなく SSE2 命令を使うことで性能が悪化することはほぼないといえる．

SSE2 命令を用いることで FP-strict のための工夫は不要となり，性能上のオーバヘッドも避けられる．しかし，SSE2 命令を持つのは 2000 年 11 月に発表された Pentium 4 以降の IA-32 プロセッサに限られる．それ以前の，1985 年に 80386 が発表されて以来の IA-32 プロセッサは SSE2 命令を持たない．SSE2 命令を持たない Pentium III などの IA-32 プロセッサも，依然，PC クラスタへの搭載といった需要があり，今後も当分の間使われ続けることは確実である．Java 仮想マシンが Pentium 4 よりも古い IA-32 プロセッサをサポートする限り，x87 命令で FP-strict な演算を効率的に達成する手法は有用である．

7. `strictfp` 導入後も残された問題

Java 言語に `strictfp` が導入されたのは，1 章で述べたように，IA-32 プロセッサが性能上のペナルティを被ることなく仕様に沿って浮動小数点演算を行えるようにするためであった．しかし実際は，ペナルティは完全に取り除かれてはいない．

`strictfp` 導入にともなう Java 言語仕様の変更とは，浮動小数点数の指数部として，IEEE 754 の単精度，倍精度の 8 ビット，11 ビットより広い幅を許容するものである[4]．`strictfp` のスコープ外では，単精度数であれば 11 ビット以上，倍精度数であれば 15 ビット以上の指数部が許されるようになった．しかし拡張倍精度が許されるようになったのは指数部だけであり，仮数部の精度は IEEE 754 単精度と倍精度のとおり，それぞれ 24 ビット，53 ビットである必要がある．

ここで，単精度数の演算が問題となる．仮数部の精度は厳密に IEEE 754 の単精度，倍精度のとおりでなければならないため，単純に，丸め精度を倍精度としたまますべての演算を行ったのでは，単精度数の演算で仮数部が言語仕様どおりの演算とは異なってしまいうる．かといって，ここで素朴に，演算に応じて丸め精度を再設定していたのでは，性能上のペナルティを被ることとなる（4.2 節）．丸め精度を倍精度のままにしておくためには，演算後にストア–リロード（3 章）を行って 24 ビット精度に丸めるという方法があるが，いずれにせよ，性能上のペナルティは避けられない．

このように，IA-32 プロセッサのハンディキャップを取り除くための言語仕様の改訂だったにもかかわらず，単精度数の演算には相変わらずペナルティがある．ただ，Streaming SIMD Extensions（SSE）命令という回避手段はある．SSE 命令群は SSE2 命令群（6 章）のサブセットである．SSE2 では 128 ビット幅の XMM レジスタで 2 つの倍精度数や 4 つの単精度数，また複数の整数を 1 度に扱えるところ，SSE で扱えるのはこのうち 4 つの単精度数のみとなっている．SSE 命令を用いることで，丸め精度の再設定やストア–リロードを行うことなく FP-strict な単精度演算を行うことが可能である．しかし，SSE 命令は Pentium III（1999 年 2 月発表）で導入された比較的新しい命令群であり，これを持たないプロセッサ，たとえば Transmeta 社 Crusoe TM5600 や Pentium II 以前の IA-32 プロセッサではこの方法はとれない．

8. ま と め

IA-32 で他の IEEE 754 準拠アーキテクチャとまったく同じ演算結果を得る手法を，Java JIT コンパイラに実装した．丸め精度は，単精度数の演算を行う場合であっても，倍精度に固定できることを示した．これにより，丸め精度を再設定するオーバヘッドを避けることができる．また，スケーリングのためには `fscale` 命令よりも乗算を用いた方が一般に効率が良いこと，そうすることで浮動小数点演算中心のプログラムで性能の低下幅を約 40%に抑えられることが分かった．最後に，Java 言語仕様に `strictfp` が導入された後も残されている IA-32 のハンディキャップを指摘した．

Java 言語と Java 仮想マシンの仕様は，実装依存をなくすこと，プラットフォームをまたいでの実行結果の再現性を高くすることなど，様々なポータビリティを強く意識して記述されている．浮動小数点演算においても同様で，演算結果の再現性を高くするための仕様の要請が厳しかったために，IA-32 は性能上のペナルティを被ることとなった．本研究は，プログラミング言語やプログラミングインタフェースの設計が，それを実装したソフトウェアの実行効率に非常に大きな影響を与えることの興味深い 1 つの事例となっている．

参 考 文 献

1) IEEE Standard 754-1985 for Binary Floating-point Arithmetic (1985).
2) Lindholm, T. and Yellin, F.: *The Java Virtual Machine Specification*, Addison Wesley (1997).
3) Gosling, J., Joy, B., Steele, G. and Bracha, G.: *Java Language Specification*, Second Edition, Addison Wesley (2000).
4) Sun Microsystems, Inc.: Updates to the Java Language Specification for JDK Release 1.2 Floating Point (1999). http://java.sun.com/docs/books/jls/strictfp-changes.pdf.
5) 首藤一幸，関口智嗣，村岡洋一：Java Just-in-Time コンパイラのためのコスト効率の良いコンパイル手法，電子情報通信学会論文誌，Vol.J86-DI, No.4, pp.217–231 (2003).
6) Shudo, K.: shuJIT—Java Just-in-Time Compiler for x86 processors (1998). http://www.shudo.net/jit/.
7) Java Grande Forum Numerics Working Group: Improving Java for Numerical Computation (1998). http://math.nist.gov//javanumerics/.
8) NaturalBridge, Inc.: BulletTrain optimizing bytecode compiler (1998). http://www.naturalbridge.com/bullettrain.html.
9) Intel Corporation: *IA–32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture* (2001).

付 録

A.1 単精度数の除算における 2 度丸め

4.2 節では，丸め精度を倍精度としたままで単精度数の四則演算を行っても 2 度丸めの問題は起きないと述べ，加算と減算，乗算についてはその理由を説明した．ここでは除算についての理由を述べる．

除算については，2 度丸めが起きないことは計算機実験を行って確認した．除数と被除数の仮数部がとりうるあらゆるビットパターンについて，2 度丸めの問題が起きうるか否かを調べた．この探索に要する時間は，部分的な探索に要した時間に基づいて，1.7 GHz Pentium 4 を 1 台用いて 1 カ月程度であると見積もることができた．実際は数台の計算機で探索空間を分担することで約 1 週間で完了した．

単精度数の仮数部は 23 ビットあり，0 から $2^{23}-1$ まで 2^{23} 通りのビットパターンをとりうる（表 1）．それぞれ 2^{23} 通りの被除数と除数，つまり 2^{46} 通りの組合せについて，商が図 7 のビットパターンに該当するか否かを調べた．これを確実に判定するには，丸め前の無限精度の演算結果を調べなければならない．しかし，計算機実験ではそれは不可能なので，ビットパターンを見て該当する可能性があるか調べ，可能性がある被除数，除数の組については，実際に除算を行うことで 2 度丸めの問題が起きるか否かを確認した．

この探索のアルゴリズムを**図 17** に示す．

▷ 0 から $2^{23}-1$ までの整数 **m** と **n** のあらゆる組合せに対して，次を行う．
- ◇ m と n からそれぞれ単精度の正規化数 f_m，f_n を作る．m，n を仮数部 23 ビットのビットパターンとし，指数部と符号は f_m と f_n で同じものを設定しておく．実際は，f_m は $+2^0(1.m)$ とした．
- ◇ $f_m \div f_n$ という除算を行い，結果を拡張倍精度（仮数部の精度 64 ビット）で得る．
- ◇ 結果の仮数部が図 7 のビットパターンに該当する可能性があるかを調べる．実際は，上位ビットから数えて 26 ビット目から 0 か 1 が 28 ビット連続していれば，可能性があると判断した．
- ◇ 該当する可能性があるなら，次を行い，実際に 2 度丸めの問題が起きるか否かを確認する．
  - ▷ e_m を 2 とし，1 から 254 までの $\mathbf{e_n}$ に対して次を行う．
    - ◇ m，n からそれぞれ単精度の正規化数 g_m，g_n を作る．m，n をそれぞれ仮数部のビットパターンとし，指数部はそれぞれ 2^{e_m-127}，2^{e_n-127} とする．
    - ◇ $g_m \div g_n$ という除算について，倍精度，単精度へと 2 度丸めた結果と，Golliver の方法で FP-strict に演算を行った結果を比較する．
    - ◇ 結果が食い違っていれば，2 度丸めの問題が起きたので，報告する．

図 17 単精度除算で 2 度丸めが起きる被除数と除数の組を探索するアルゴリズム
Fig. 17 Algorithm to search pairs of dividend and divisor which suffer double-rounding in single-precision division.

表 4 除数の指数部 e_n と演算結果の関係
Table 4 Quotients corresponding to exponent of divisor e_n.

e_n	演算結果	結果の種別
1	$2^1 \cdot 1.01$	正規化数
2	$2^0 \cdot 1.01$	正規化数
	...	
128	$2^{-126} \cdot 1.01$	正規化数
129	$2^{-126} \cdot 0.101$	非正規化数
130	$2^{-126} \cdot 0.0101$	非正規化数
131	$2^{-126} \cdot 0.00101$	非正規化数
	...	
150	$2^{-126} \cdot 0.$(0 が 21 ビット連続)10	非正規化数
151	$2^{-126} \cdot 0.$(0 が 22 ビット連続)1	非正規化数
152	$2^{-126} \cdot 0.$(0 が 22 ビット連続)1	非正規化数
153	0	ゼロ
154	0	ゼロ
	...	

実際の除算で 2 度丸め問題を確認する際に，被除数の仮数部 e_n を 1 から 254 まで試すのは，図 7 の 2 度目の丸め位置を網羅するためである．丸め前の無限精度の仮数部は，ビットパターンは指数部の影響を受けず一定なのだが，この，仮数部上の 2 度目の丸め位置は指数部の影響を受ける．どの位置で 2 度目の丸めが起きても 2 度丸め問題が起きないことを確認する必要があるため，様々な e_n で確認するのである．たとえば，m と n から作成した仮数部がそれぞれ 1.01，1.0 の場合，それぞれの e_n に対応する除算の結果は**表 4** に示すとおりである．この場合，e_n が 128 までは単精度の正規化数として表せるため，仮数部上の 2 度目の丸め位置は一定である．e_n が 129 以上となると非正規化数となり，図 7 でいうところの 2 度目の丸め位置は e_n に応じて変わる．そのため，この場合であれば，結果が正規化数となるような指数に加えて，結果が非正規化数となる 129 以上の e_n についても調べる必要があるのである．1 から 254 までの e_n について調べるのは無駄ではあるが，網羅するためには十分な範囲である．

上述のアルゴリズムでは，被除数 f_m，g_m，除数 f_n，g_n として正規化数のみを考えている．それでも，非正規化数の仮数部がとるビットパターンも網羅できるため，正規化数のみを考えれば十分だからである．たとえば，$2^{-126} \cdot 0.011 \div 2^{-126} \cdot 0.00101$ という非正規化数どうしの除算について考える代わりには，$2^0 \cdot 1.1 \div 2^{-1} \cdot 1.01$ という正規化数の除算について考えれば済む．

(平成 14 年 4 月 30 日受付)
(平成 15 年 4 月 3 日採録)

首藤 一幸（正会員）

1996 年早稲田大学理工学部情報学科卒業．1998 年同大学メディアネットワークセンター助手．2001 年同大学大学院理工学研究科情報科学専攻博士後期課程修了．同年産業技術総合研究所入所．現在に至る．博士（情報科学）．分散処理方式，プログラミング言語，言語処理系，情報セキュリティ等に興味を持つ．IEEE–CS，ACM 各会員．

関口　智嗣（正会員）

1984年工業技術院電子技術総合研究所入所．以来，データ駆動型スーパーコンピュータSIGMA-1の開発，ネットワーク数値ライブラリNinf，クラスタコンピューティング，グリッドコンピューティング等に関する研究に従事．2001年独立行政法人産業技術総合研究所に改組．2002年1月より同所グリッド研究センター長．市村賞，情報処理学会論文賞受賞．グリッド協議会会長．日本応用数理学会，日本計算工学会，SIAM，IEEE，つくばサイエンスアカデミー各会員．

村岡　洋一（正会員）

1965年早稲田大学理工学部電気通信学科卒業．1971年イリノイ大学電子計算機学科博士課程修了．Ph.D.この間，Illiac-IVプロジェクトで並列処理ソフトウェアの研究に従事．同学科助手ののち，日本電信電話公社（現NTT）電気通信研究所に入所．1985年より早稲田大学理工学部教授．現在同大学メディアネットワークセンター所長．並列処理，マンマシンインタフェース等に興味を持つ．「コンピュータアーキテクチャ」（近代科学社）等著書多数．