浮動小数点演算の実装メモ

(revision 3)

安達 知也* (Team Sakanaya)

平成 20 年 12 月 16 日

1 はじめに

1.1 この文書について

この文書は、2006 年度東京大学理学部情報科学科「情報科学実験 II(CPU 実験)」に参加した Team Sakanaya の FPU(およびライブラリ) について記したものです。実装の詳細を示し、誤差評価を行うことで、実験の浮動小数点演算実装基準 [1] を (ほぼ) 完全に満たしていることを主張します。

1.2 実験環境

実験に使った FPGA、および回路の合成ツールは以下のとおりです。

FPGA	Xilinx Virtex-II (XC2V1000-4FF896)
合成ツール	Xilinx ISE 6.3.03i(単体テスト時) / 7.1.04i(CPU に組み込み時)

2 数値表現

2.1 実装基準

IEEE 規格の+0 とノーマル数は最低限表現できなければならず、これらの浮動小数点値に対しては対応する実数値が定義される。それ以外の浮動小数点値の定義/未定義および値の解釈は指定しないが、個々の浮動小数点値について対応する実数値が定義されているか否かを一意に定めねばならない。

2.2 FPU 上の実装

基本的に IEEE754 の 32bit 単精度浮動小数点数の表現と同じですが、以下の点が異なります。

- 指数部が 0 のものは全てゼロ
- 指数部が 255 のものも正規化数として扱う

こうすることで、オーバーフロー処理が不要になり、アンダーフロー処理で仮数部を zero fill する手間も省くことができます。

*adachi@il.is.s.u-tokyo.ac.jp

3 ハードウェア実装

3.1 FADD/FSUB

3.1.1 実装基準と解釈

有効な浮動小数点数 A, B の組が $-2^{127} < A, B, A \circ B < 2^{127}$ を満たすとき $\texttt{FADD}(A, B)$ および $\texttt{FSUB}(A, B)$ は有効な浮動小数点数を返し、以下の条件を満たすこと。($\epsilon = 2^{-126}$)

$$|\texttt{FADD}(A, B) - (A + B)| < \max(|A| \cdot 2^{-23}, |B| \cdot 2^{-23}, |A + B| \cdot 2^{-23}, \epsilon)$$

$$|\texttt{FSUB}(A, B) - (A - B)| < \max(|A| \cdot 2^{-23}, |B| \cdot 2^{-23}, |A - B| \cdot 2^{-23}, \epsilon)$$

「± 何 *ulp* の誤差を許す」ではなく、「round bit の OR 計算や丸めをしなくてよい」という意味になります。簡単に説明すると、

- 次節 Step 2. に関して
  $|A| > |B|$ とすると、B を右シフトしてはみ出る部分は、$|A| \cdot 2^{-23}$ 未満の大きさしか持っていないので、それを無視しても数学的な演算との差は $|A| \cdot 2^{-23}$ 未満になります。$|A| < |B|$ についても同様です。

- 次節 Step 6. に関して
  丸めは guard bit が 1 のときしか起きないことから、丸めで増える量は $0.5ulp$ 以下になります。よって、丸めをサボることによって生じる誤差は $|A \circ B| \cdot 2^{-24}$ 以下です。

ただし、実装時に (正確にはこの文書を書く際に再検証を行うまで) 基準を誤解していたため、設計した FPU は右シフト時の処理や丸めを行った実装になっています。基準の誤解により、「過去の班の実装は基準を満たしていない」と半ば馬鹿にするような発言を、口頭および班 wiki 上で行ったことは申し訳なく思っています。$1.0 * 2^n - 1.11 \ldots 11 * 2^{n-1}$ の演算結果が指数部までズレるのはさすがにどうかと思いますが、基準を満たしていることに間違いはありません。この場を借りて謝罪いたします。

3.1.2 演算手順

おおまかな演算手順は、以下のとおりになります。

Step 1. 二数の絶対値の大きさを比べる

Step 2. 小さいほうの数の仮数部を、指数部の差だけ右シフトする

Step 3. 二つの数の符号部と、指定された演算とを加味して、実際に行う演算を決定する

Step 4. その演算を行う

Step 5. 正規化数になるようにシフトを行い、それに合わせて指数部も適宜加減する

Step 6. 丸め処理を行う (このとき、指数部に 1 繰り上がることがある)

「小林先生の本」[2] 等の教科書で触れられているとおり、Step 2. の右シフト時には $1/2ulp$、$1/4ulp$ の bit にくわえて、$1/8ulp$ 以下の bit 全てを OR したものも保存しておく必要があります。つまり、Step 4. の演算は、(加算時の繰り上がりに対応するための 1bit) + (仮数部の上にある暗黙の '1') + (仮数部 23bit) + (丸め処理用の 3bit) の 28bit で行うことになります。

上記手順に従えば、正規化数の演算に関しては市販の (まともな)FPU と同じ結果が出ることが期待されます。数学的な演算との誤差は $|A \circ B| \cdot 2^{-24}$ 以下になります。

3.1.3 FPGA への実装

100MHz 3clock で実装した際の各 stage の詳細は以下のとおりです。

Stage1

1. 二数の絶対値の大きさを比較 (Step 1.)
2. 右シフト量を計算し、二数の仮数部をシフトしておく
   (入力が 0.0 だった場合はゼロクリア)
3. 1. の結果を使って、「大きい方」を求める
4. 1. の結果で 2. の結果を Mux して「小さい方をシフトしたもの」を求める (Step 2.)
5. 「大きい方」の指数部が答えの指数部の元になるので保持しておく
6. 「大きい方」の符号部が答えの符号部になるので保持しておく
7. 実際に行う演算の決定 (Step 3.)

制御部からの信号線の遅延があり、さらに 31bit の比較をする必要があるので、あまり多くのことができません。右シフト演算は round bit の OR をしないといけないので、自前でモジュールを用意しました。

Stage2

1. 「大きい方」と「小さい方をシフトしたもの」について演算を行う (Step 4.)
2. 1. の結果を Zero Leading Counter[1]にかける
3. 丸めによって指数部に繰り上がりが発生するか否かを判定
4. 2. の下位 2bit を使って一回目の左シフトを行う
5. 2. の下位 2bit を使って丸め項を決定
6. 3. の結果を使って指数部に補正を加える
7. 符号部はそのまま

[1] ぐぐっても 4 件しかかからないので一般的な用語ではないようですが、これで慣れてしまったのでこの文書中では ZLC と呼びます。本当は leading-zero counter のほうが正しいはず。要は先頭のゼロの数を数える Priority Encoder。

28bit の加減算だけでそれなりの時間がかかる上に、ZLC はその性質上入力が全て揃わないと答えが決定しないため、あまり他のことをする余裕がありません。ただ、Stage3 も同じくらい余裕がないので、シフト演算を二つに分けて、その一部をこちらで行います。丸めが発生するのは ZLC の出力が 0〜2 のときだけなので、丸め項もあわせて計算します。

なお、ZLC は 28bit 分用意する必要はありません。1. の結果で一番小さいものは $00\ldots00$(先頭から 0 が 28 個連続) ですが、その次に小さいのは $00\ldots00100$(先頭から 0 が 25 個連続) なので、先頭 26bit だけ見てやればゼロかどうかを判定することができます。

また、1. の演算結果が $011\ldots11XX$ あるいは $001\ldots111X$ だった場合 (1 が 25 個以上連続)、仮数部に丸め項を足すと繰り上がりが発生して指数部が変わりますが、それを愚直にやっていると Stage3 のタイミングが厳しくなるので、Stage2 のうちにそれを見越して指数部に補正をかけてしまいます。

Stage3

1. ZLC の出力の上位 3bit が 0 なら丸め項加算、それ以外の場合は左シフト (Step 5,6.)
2. 指数部から ZLC の出力を引く (Step 5.)
3. 2. の結果が負、あるいは ZLC の出力が 26 の場合、ゼロを返すために指数部をゼロクリア
4. 符号部と指数部と仮数部をくっつけて返す

23bit の carry chain の後にシフトによる Mux が入り、さらに制御部まで値を伝えなければならないので、かなり hard です。最初に何も考えずに実装したときには「20ns ぐらいかかる」と ISE に言われて絶望した覚えがあります。Stage2 に追いやれるものを徹底的に追いやったり、ゼロの表現を変えて仮数部をゼロクリアしなくて済むようにしたりして、なんとかまともに動くものを作ることができました。

3.1.4 誤差評価

教科書通りの丸めを行っているため、理論的に誤差は $|A \circ B| \cdot 2^{-24}$ 以下に抑えられているはずです。また、C によるエミュレーションを行い、10 億回の乱数テストおよび考えうるコーナーケースのテスト全てで x87 の fadd/fsub 命令と答えが一致することを確認しました。全数チェックは探索空間が単純計算で 2^{64} になってしまうため無理だと判断しました。

3.2 FMUL

3.2.1 実装基準と解釈

有効な浮動小数点数 A, B の組が $-2^{127} < A, B, AB < 2^{127}$ を満たすとき $\texttt{FMUL}(A, B)$ は有効な浮動小数点数を返し、以下の条件を満たすこと。($\epsilon = 2^{-126}$)

$$|\texttt{FMUL}(A, B) - AB| < \max(|AB| \cdot 2^{-22}, \epsilon)$$

十分条件は「$\pm 2ulp$ 未満の誤差」。答えが 2 ベキでない場合は、$\pm 2ulp$(以上) ズレていていも平気です。

3.2.2 演算手順

おおまかな演算手順は、以下のとおりになります。

Step 1. 二数の仮数部を上位 12bit と下位 11bit に分ける

Step 2. 暗黙の '1' をつけて、13bit*13bit(HH)、13bit*11bit(HL)、11bit*13bit(LH) の計算を行う

Step 3. $HH + (HL >> 11) + (LH >> 11) + 2$ を計算する

Step 4. 丸めは行わず、最上位の '1' から下 23bit を答えの仮数部とする

Step 5. 指数部は足し算、符号部は XOR

組み込みの乗算器が 18bit*18bit しか計算できないので、上位と下位に分けて計算します。

3.2.3 FPGA への実装

100MHz 3clock で実装した際の各 stage の詳細は以下のとおりです。

Stage1

1. HH、HL、LH を計算 (Step 2.)
2. $exp_1 + exp_2 + 129$ で指数部を計算 (Step 5.)
3. 符号部を XOR (Step 5.)

急いでも 2clock には収まりそうにないので、のんびりと計算します。2. で 129 を足しているのは、指数部に 127 のオフセットがかかっているためです。8bit ではなく 9bit の足し算で、最上位 bit がアンダーフロー検出の役割を果たします。指数部が 0 の場合は、アンダーフローしたことにして最後に 0 で埋めてもらいます。

Stage2

1. $HH + (HL >> 11) + (LH >> 11) + 2$ を計算 (Step 3.)
2. 指数部に 1 足したものを計算しておく

足し算は項数が多いので時間がかかるようです。アンダーフロー時に指数部を 0 で埋めるのはここでやっておいてもよかったかもしれません。

Stage3

1. underflow bit と仮数部の最上位 bit を見て、指数部を選ぶ
2. 仮数部を正規化 (Step 4.)
3. 符号部と指数部と仮数部をくっつけて返す

31bit の Mux が入るため、大きな回路になります。ちなみに、オーバーフロー時にもアンダーフローと判定されます。8bit 目も見れば簡単に判別可能ですが。

3.2.4 誤差評価

演算手順の Step 4,5. の手順で目的の誤差内に収まることを示します。24bit*24bit=48bit で計算したものと、答えの仮数部との差がどれほどになるかを求めます。

まず、Step 4. の誤差を見積もります。簡便のため、小数点の位置を気にせず、48bit 整数だと思って議論します。HL の下位 11bit、LH の下位 11bit、11bit*11bit(LL) を omit することによって生じる最大誤差は、

$$11\ldots11 * 2^{11} + 11\ldots11 * 2^{11} + 11\ldots11 * 11\ldots11 = 101111111110000000000001$$

となります。最小誤差は 0 です。「+2」の項を加味すると、この項の大きさは 2^{23} なので、

$$-11111111100000000001 \leq |(\text{Step 4.})| - |AB| \leq +2^{23}$$

が成り立ちます。

また、切り捨てることによる誤差は、$-1ulp < |(\text{切り捨て後})| - |(\text{切り捨て前})| \leq 0$ になります。

これら二つを合わせて、Step 4,5. の誤差を見積もります。ulp の場所が二通り考えられるので、場合分けします。

1. 最上位 bit(48bit 目) が立っている場合
   この場合、ulp は 25bit 目なので、Step 4. の誤差は $-1/4ulp < |(\text{Step 4.})| - |AB| \leq +1/2ulp$ になります。切り捨てまで考えると、$-5/4ulp < |(\text{Step 4,5.})| - |AB| \leq +1/4ulp$ となり、基準を満たします。

2. 最上位 bit(48bit 目) が立っていない場合
   47bit 目が立っていることは自明なので、ulp は 24bit 目にあります。このとき、Step 4. の誤差は $-1/2ulp < |(\text{Step 4.})| - |AB| \leq +1ulp$ です。$-3/2ulp < |(\text{Step 4,5.})| - |AB| \leq +1ulp$ となり、基準を満たします。

また、C によるエミュレーションを行い、10 億回の乱数テスト全てで x87 の fmul 命令との差が $\pm 1ulp$ に収まることを確認しました。全数チェックは探索空間が単純計算で 2^{64} になってしまうため FADD 同様無理だと判断しました。

3.3 FINV/FSQRT

3.3.1 実装基準と解釈

有効な浮動小数点数 $A, B(B \neq 0)$ の組が $-2^{127} < A, B, \frac{A}{B} < 2^{127}$ を満たすとき FDIV(A, B) は有効な浮動小数点数を返し、以下の条件を満たすこと。($\epsilon = 2^{-126}$)

$$\left|\texttt{FDIV}(A, B) - \frac{A}{B}\right| < \max\left(\left|\frac{A}{B}\right| \cdot 2^{-20}, \epsilon\right)$$

有効な浮動小数点数 A が $0 \leq A < 2^{127}$ を満たすとき FSQRT(A) は有効な浮動小数点数を返し、以下の条件を満たすこと。($\epsilon = 2^{-126}$)

$$\left|\texttt{FSQRT}(A) - \sqrt{A}\right| < \max\left(\sqrt{A} \cdot 2^{-20}, \epsilon\right)$$

両方とも十分条件は「$\pm 8ulp$ 未満の誤差」。FDIV を FINV と FMUL を組み合わせて (ソフトウェア的に) 作ることを考えると、FINV は「$\pm 4ulp$ 未満の誤差」でよさそうです。

3.3.2 演算手順

おおまかな演算手順は、以下のとおりになります。

Step 1. 線型近似テーブルを引く

(a) FINV の場合、仮数部の上位 10bit をキーにして

(b) FSQRT の場合、指数部の最下位 bit と仮数部の上位 9bit をキーにして

Step 2. 下位 13〜14bit を使って ax+b を計算

Step 3. 指数部は適宜計算、符号部はそのまま

3.3.3 テーブル生成

FINV

基本はニュートン法を使って、初期値 x_0 に対して、$\mathtt{FINV}(A) = 2x_0 - Ax_0{}^2$ で計算します。仮数部だけに着目すると、24bit 整数 A, x_0(暗黙の '1'+仮数部) に対して、$finvman(A) = 2x_0 - Ax_0{}^2/2^{46}$ となります。ここで、x_0 が A の上位 $k$bit をキーにしてテーブル引きする値であることに注意して、$A = A_0 \cdot 2^{23-k} + A_1$ と表すと、$finvman(A) = 2x_0 - (A_0 \cdot 2^{23-k} + A_1){x_0}^2/2^{46} = (2x_0 - A_0{x_0}^2/2^{23+k}) - A_1{x_0}^2/2^{46}$ となり、前半が定数項、後半が (A_1 の) 一次の項の、線型近似が得られたことになります。なお、初期値は $1/(A_0 \cdot 2^k)$ と $1/((A_0+1) \cdot 2^k)$ の平均値を用いました。また、実際にはテーブルに使う BlockRAM のバンド幅の関係からフルのデータは格納できないので、定数項を 23bit、勾配を 13bit に丸めます。

それだけでは誤差が大きいので、パラメータの調整を行います。$0 \leq A_1 < 2^k$ の範囲において、x87 演算による結果との差の分布をとり、

- 分散が最小、かつ
- k *ulp* の差に対して k^2 のペナルティを与えたときの総スコアが最小

となるようにパラメータを微調整しました。(分散は定数項に影響されないことに注意)

FSQRT

FINV と同じで、ニュートン法の式 $\mathtt{FSQRT}(A) = x_0/2 + A/(2x_0)$ から線型近似を導き、パラメータの調整をします。ただし、指数部が奇数のときと偶数のときで式が変わるので注意が必要です。

3.3.4 FPGA への実装

100MHz 3clock で実装した際の各 stage の構成は以下のとおりです。

Stage1

1. BlockRAM に渡すアドレスを計算

アドレス計算と書いていますが、ただの Mux です。制御部からの距離が遠いので、余裕を持たせています。

Stage2

1. BlockRAM から値が出てくる
2. 乗算器で仮数部の下位 13〜14bit に勾配を掛ける

元の式で計算すると乗算器が複数必要ですが、式を簡単にしたことで乗算器が一つで済んでいます。

Stage3

1. 減算 (`FINV`) または加算 (`FSQRT`) を行って、仮数部をつくる
2. 指数部を計算する (ゼロのときはゼロを返す)
3. 符号部はそのまま返す

一番忙しい Stage です。でも、特に書くことはありません。$\texttt{FINV}(0) = 0$ は素敵だと思います。

3.3.5 誤差評価

理論ではどうしようもないので、実際に C によるエミュレーションで全数チェックして評価します。

`FINV`
仮数部 23bit を動かして x87 演算との差を見ます。x87 演算が正しいという仮定の下では、$|(\text{x87 演算}) - A/B| \leq 0.5ulp$ が成り立つはずなので、x87 演算との差が $\pm 3ulp$ に収まっていれば十分です。
結果は以下のとおりです。

-3	12539
-2	192340
-1	1489148
0	4691652
$+1$	1914451
$+2$	88171
$+3$	307
sum	8388608

`FSQRT`
指数部最下位+仮数部の 24bit を動かして x87 演算との差を見ます。x87 演算が正しいという仮定の下では、$|(\text{x87 演算}) - A/B| \leq 0.5ulp$ が成り立つはずなので、x87 演算との差が $\pm 7ulp$ に収まっていれば十分です。
結果は以下のとおりです。

-1	2748131
0	11385094
$+1$	2615519
$+2$	28472
sum	16777216

4 ソフトウェア実装

4.1 SIN/COS

4.1.1 実装基準と解釈

ある定数 c $(1-2^{-23} < c < 1+2^{-23})$ が存在して、$-2^{127} < A < 2^{127}$ を満たす全ての有効な浮動小数点数 A に対して SIN(A) および COS(A) は有効な浮動小数点数を返し、以下の条件を満たすこと。$(\epsilon = 2^{-126})$

$$|\mathtt{SIN}(A) - \sin cA| < \max(|\sin cA| \cdot 2^{-18}, \epsilon)$$

$$|\mathtt{COS}(A) - \cos cA| < \max(|\cos cA| \cdot 2^{-18}, \epsilon)$$

c が曲者ですが、これは数学的な π と「浮動小数点数で近似した π」(以下 PI) の比を表すものだと解釈できます。SIN, COS の周期を 2π ではなく 2PI にすることで、「全ての有効な浮動小数点数」から $[0, 2\mathtt{PI})$ への写像 (reduction) を簡単に書くことができます。要求精度に関しては、この c が右辺に入っているため、単純に「$\pm 32ulp$ 未満の誤差」と言い切ることはできません。

4.1.2 実装概要

大きく分けて、以下の四段階に分かれます。

preparation
簡便のため、絶対値をとります。SIN の場合、落とした符号部を FLAG とします。COS の場合、FLAG $= +$ とします。FLAG は最終的に答えの符号部になります。

reduction I
SIN, COS が 2PI 周期の関数であることを利用して、「全ての有効な浮動小数点数」から $[0, 2\mathtt{PI})$ への reduction を行います。要は 2PI で割った余りをとればよいのですが、単純に割り算すると誤差が出るので、exact な計算を行います。「浮動小数点数 x, y $(x > 2^{-100})$ について、$y \le x \le 2y$ のとき、$x - y$ に exact に対応する浮動小数点数 z が存在する[2]」ことを利用します。

reduction II
SIN と COS が対になっていることを利用して、$[0, 2\mathtt{PI})$ から $[0, \mathtt{PI}/4]$ への reduction を行います。以下の手順を 1. から順に行い、最終的に呼ぶ関数を決定します。

1. $A \ge$ PI のとき、$A = A -$ PI として、FLAG を反転
2. $A \ge$ PI$/2$ のとき、$A =$ PI $- A$ として、COS が呼ばれていた場合は FLAG を反転
3. $A \le$ PI$/4$ のとき、SIN なら $kernel\_sin$、COS なら $kernel\_cos$ を呼ぶ
   $A >$ PI$/4$ のとき、$A =$ PI$/2 - A$ として、SIN なら $kernel\_cos$、COS なら $kernel\_sin$ を呼ぶ

kernel
Taylor 展開ベースの多項式 $kernel\_sin, kernel\_cos$ で計算します。最後に符号部に FLAG をつけ加えて、値を返します。

[2]証明略。FADD, FSUB が右シフト時に $1ulp$ 未満を切り捨てる仕様になっていると成り立ちません。-100 は適当ですが、$0 < x - y < 2^{-126}$ になることがあるとマズいので。

4.1.3 パラメータ選択

kernel_sin の多項式は、[3] に載っているものを利用しました。x87 の fsin 命令の結果との差が $\pm 1ulp$ に収まったため、特に変更は加えませんでした。

kernel_cos の多項式は [3] では明記されておらず、関連検索でたどりついた [4] に載っている式もあまりいい挙動を示さなかったため、自力で決定しました。Taylor 展開を 6 次の項で打ち切り、4 次の項と 6 次の項の係数を変化させ、x87 の fcos 命令の結果との差が $\pm 3ulp$ に収まるようにしました。

具体的な関数は以下のとおりです。

$$kernel\_sin(A) = A - 0.16666668\ (\mathrm{0xbe2aaaac}) * A^3 + 0.008332824\ (\mathrm{0x3c088666}) * A^5 \\ -0.00019587841\ (\mathrm{0xb94d64b6}) * A^7$$

$$kernel\_cos(A) = 1.0 - 0.5 * A^2 + 0.04166368\ (\mathrm{0x3d2aa789}) * A^4 - 0.0013695068\ (\mathrm{0xbab38106}) * A^6$$

4.1.4 擬似コード

```
fsin(A){
  FLAG = flag(A);
  abs(A);
  reduction_2pi(A);
  if(A >= PI){
    A = A - PI;
    reverse(FLAG);
  }
  if(A >= PI/2){
    A = PI - A;
  }
  if(A <= PI/4){
    kernel_sin(A);
  }else{
    A = PI/2 - A;
    kernel_cos(A);
  }
  add FLAG;
}
```

```
fcos(A){
  FLAG = '+';
  abs(A);
  reduction_2pi(A);
  if(A >= PI){
    A = A - PI;
    reverse(FLAG);
  }
  if(A >= PI/2){
    A = PI - A;
    reverse(FLAG);
  }
  if(A <= PI/4){
    kernel_cos(A);
  }else{
    A = PI/2 - A;
    kernel_sin(A);
  }
  add FLAG;
}
```

```
reduction_2pi(A){
  P = PI * 2;
  while(A >= P){
    P = P * 2;
  }
  while(A >= PI * 2){
    if(A >= P){
      A = A - P;
    }
    P = P / 2;
  }
}

kernel_sin(A){
  A - S3 * A^3
    + S5 * A^5 - S7 * A^7;
}

kernel_cos(A){
  1.0 - C2 * A^2
    + C4 * A^4 - C6 * A^6;
}
```

4.1.5 誤差評価

まず、c が π と PI の比であることから、$c = \pi/\mathtt{PI}$ と書けます。$A = 2n\mathtt{PI} + A_0$ ($n \in \mathbf{Z},\ 0 \le A_0 < 2\mathtt{PI}$) と書けるとき[3]、$cA = 2n\pi + cA_0$ なので、$\mathtt{SIN}(A) = \mathtt{SIN}(A_0)$、および $\sin cA = \sin cA_0$ が成り立ちます。$|\mathtt{SIN}(A) - \sin cA| < \max(|\sin cA| \cdot 2^{-18}, \epsilon) \Longleftrightarrow |\mathtt{SIN}(A_0) - \sin cA_0| < \max(|\sin cA_0| \cdot 2^{-18}, \epsilon)$ となるので、$0 \le A < 2\mathtt{PI}$ について誤差評価ができれば十分です。COS についても同様です。

同様に、reduction II で行った $[0, 2\mathtt{PI}) \longrightarrow [0, \mathtt{PI}/4]$ の操作も、c を介して $\sin, \cos, \pi$ の世界に直せば単なる加法定理にすぎないので、$0 \le A \le \mathtt{PI}/4$ についての誤差評価で十分なことがわかります。($A = n\mathtt{PI}/4 + A_0$ ($n = 0, 1, \ldots, 7,\ 0 \le A_0 < \mathtt{PI}/4$) とすれば式変形で説明可能)

[3] A_0 に exact に一致する浮動小数点数が存在することは明らかです。

まず、$A \leq 2^{-126}$ の場合について評価します。$A = 0$ の場合は、答えが 0 になるので OK です。$A = 2^{126}$ の場合は、ϵ の項が効いてくるので OK です。(reduction の結果 $A = 2^{126}$ になることはない)

残りの場合については、実際にエミュレーションして評価しますが、その前に、$\sin cA$ と $\cos cA$ の大きさを x87 演算の結果を使って評価しておきます。PI = 3.14159274101257324219 75 (0x40490fdb) とすると、$1 - 2^{-25} < c = \pi/\mathtt{PI} < 1$ となります。一方、A より小さい浮動小数点数の中で一番大きい浮動小数点数 B を考えると、$B <= (1 - 2^{-24})A$ なので、$B < cA < A$ が得られます。sin は $[0, \mathtt{PI}/4]$ では単調増加なので、$\sin B < \sin cA < \sin A$ が成り立ちます。これを使って $\sin cA$ を評価すると、x87 の fsin 命令が正確な丸めを行っているという仮定の下で、$x87sin(B) - 0.5ulp \leq \sin cA \leq x87sin(A) + 0.5ulp$ がいえます。さらに、実験事実として $(2^{-126}, \mathtt{PI}/4]$ では $x87sin(A) - x87sin(B) \leq 2ulp$ であることがわかるので、$x87sin(A) - 2.5ulp \leq \sin cA \leq x87sin(A) + 0.5ulp$ として評価すればよいことになります。

同様にして $\cos cA$ についても評価すると、$[0, \mathtt{PI}/4]$ で $x87cos(B) - x87cos(A) \leq 1ulp$ となるので、$x87cos(A) - 0.5ulp \leq \cos cA \leq x87cos(A) + 1.5ulp$ が得られます。

[0, PI/4] において SIN と $x87sin$、COS と $x87cos$ の差をそれぞれとった結果を以下に示します。

	SIN	COS
-3		16888
-2		115018
-1	960588	1752835
0	1060571055	1055607911
$+1$	221153	1715615
$+2$		1886308
$+3$		658221
sum	1061752796	1061752796

以上から、$\mathtt{SIN}(A), \mathtt{COS}(A)$ と $\sin cA, \cos cA$ の差が $5ulp$ 未満に収まっていることがわかります。$\pi/\mathtt{PI}$ の評価をもう少し正確に行い、sin や cos が上に凸であることを利用して $\sin cA, \cos cA$ の値をさらに厳密に抑えることができれば、$4ulp$ 未満であることが示せる気もしますが、許容誤差が $32ulp$ なので、この程度の説明にとどめておきます。

4.2 ATAN

4.2.1 実装基準と解釈

$-2^{127} < A < 2^{127}$ を満たす全ての有効な浮動小数点数 A に対して $\mathtt{ATAN}(A)$ は有効な浮動小数点数を返し、以下の条件を満たすこと。($\epsilon = 2^{-126}$)

$$|\mathtt{ATAN}(A) - \arctan A| < \max(|\arctan A| \cdot 2^{-20}, \epsilon)$$

十分条件は「$\pm 8ulp$ 未満の誤差」。

4.2.2 実装概要

Taylor 展開ベースの多項式 $kernel\_atan$ を使います。ただし、$kernel\_atan$ は $|A|$ が大きくなればなるほど精度が悪くなるので、絶対値が大きな数に対しては、加法定理を使って reduction を行います。具体的には、以下の三通りに場合分けします。

- $|A| < 0.4375$ の場合
  $kernel\_atan(A)$ を計算します。

- $0.4375 \leq |A| < 2.4375$ の場合
  $PI/4 + kernel\_atan((|A| - 1.0)/(|A| + 1.0))$ を計算し、符号部のつじつまを合わせます。

- $2.4375 < |A|$ の場合
  $PI/2 - kernel\_atan(1/|A|)$ を計算し、符号部のつじつまを合わせます。

4.2.3 パラメータ選択

$kernel\_atan$ の多項式は、Taylor 展開を 13 次の項で打ち切ったものをベースにしました。11 次の項と 13 次の項の係数を変化させ、x87 の fpatan 命令の結果との差が $\pm 4ulp$ に収まるようにしました。具体的な関数は以下のとおりです。

$$\begin{aligned} kernel\_atan(A) = A &- 0.3333333\ (\text{0xbeaaaaaa}) * A^3 + 0.2\ (\text{0x3e4ccccd}) * A^5 \\ &- 0.142857142\ (\text{0xbe124925}) * A^7 + 0.111111104\ (\text{0x3de38e38}) * A^9 \\ &- 0.08976446\ (\text{0xbdb7d66e}) * A^{11} + 0.060035485\ (\text{0x3d75e7c5}) * A^{13} \end{aligned}$$

4.2.4 誤差評価

C によるエミュレーションで正の数について全数チェックを行い、x87 の fpatan 命令との差の統計をとりました。以下にその結果を示します[4]。

-3	7
-2	15427
-1	516258
0	2038740991
$+1$	90952291
$+2$	439267
$+3$	41564
$+4$	627
sum	2130706432

4.3 ITOF

4.3.1 実装基準と解釈

> 32bit 整数 I に対して $\texttt{ITOF}(I)$ は有効な浮動小数点数を返す。このとき、$|A - I| < |\texttt{ITOF}(I) - I|$ を満たすような有効な浮動小数点数 A が存在しないこと。

$|I| \leq 2^{24} = 16777216$ のときは、exact に対応する浮動小数点数が存在するので、その値を返します。$|I| > 16777216$ のときは、exact に対応する浮動小数点数が存在しないことがあるので、適当に丸めを行って一番近い浮動小数点数を返します。「一番近い」ものが二つ存在する場合は、どちらを返してもよい (round-to-even にこだわらなくてよい) と解釈できます。

[4] 和が 2147483648 でないのは、$A \geq 2^{127}$ となる [0x7f000000,0x7fffffff] を除いたため。

4.3.2 実装概要

$|I| < 2^{23} = 8388608$ と $|I| \geq 8388608$ で場合分けします。ITOF は奇関数なので、以下では $I \geq 0$ として議論します。

$I < 8388608$ のとき

$f(I) = I + 0\text{x}4\text{b}000000$ とします。0x4b000000 は浮動小数点数として見ると 8388608.0 です。$1ulp$ の大きさがちょうど 1.0 なので、仮数部に 1 を足すと、1.0 増えます。つまり、$f(I)$ は $[0, 8388608)$ から $[8388608.0, 16777216.0)$(浮動小数点数としてみたとき) への写像となります。あとは FSUB で 8388608.0 を引いてやれば、答えの浮動小数点数が得られます。

$I \geq 8388608$ のとき

この場合は I が大きくて指数部まで食い込んでしまうので、上で使った手法はそのままでは使えません。$I = m \cdot 8388608 + n$ $(m \in \mathbf{N},\ 0 \leq n < 8388608)$ として、ITOF$(m \cdot 8388608)$ と ITOF(n) を別々に計算します。

ITOF$(m \cdot 8388608)$ は、8388608.0 を m 回足して求めます。8388608.0 は仮数部が全て 0 なので、m 回の FADD 程度では情報落ち等の誤差は発生しません。ITOF(n) は、上で使った手法で求めることができます。

最後に、これら二つを FADD します。このとき発生する情報落ちは、「exact に対応する浮動小数点数が存在しない」ことに起因するものなので、仕方ありません。FADD の中で丸めが行われて、「一番近い」浮動小数点数を返すことができます。

4.3.3 誤差評価

丸め処理を FADD に任せているので、FADD が OK ならばこちらも大丈夫です。一応 C によるエミュレーションで全数チェックを行って、x87 の fild 命令の結果と完全に一致することを確かめました。

4.4 FTOI

4.4.1 実装基準と解釈

> 有効な浮動小数点数 A が $-2^{31} + 1 \leq A \leq 2^{31} - 1$ を満たすとき、FTOI(A) は 32bit 整数を返す。
> このとき、$|I - A| < |\text{FTOI}(A) - A|$ を満たすような 32bit 整数 I が存在しないこと。

$|A| \geq 2^{31}$ の場合の返り値については明記されていないので、特に何か特別な処理をしてやる必要はありません。ITOF と同じく、「一番近い」ものが二つ存在する場合は、どちらを返してもよい (round-to-even にこだわらなくてよい) と解釈できます。

4.4.2 実装概要

ITOF の逆関数 (のようなもの) なので、ITOF の手順を逆にたどります。FTOI も奇関数なので、以下では $A \geq 0.0$ として議論します。

$A < 8388608.0$ **のとき**
FADD で 8388608.0 を足してから、整数演算の sub で 0x4b000000 を引いてやれば、求める答えが得られます。丸めは FADD の中で行われるので、勝手に round-to-even になっています。

$A \geq 8388608.0$ **のとき**
ITOF 同様、$A = m \cdot 8388608.0 + A_0$ ($m \in \mathbf{N}$, $0.0 \leq A_0 < 8388608.0$) として二つに分けてから、後で足し合わせます。m は A から 8388608.0 を引いた回数として求めます。$A < 2^{31}$ なので、8388608.0 を引く際に情報落ちが発生することはありません。

4.4.3 誤差評価

ITOF 同様、丸め処理を FADD に任せているので、FADD が OK ならばこちらも大丈夫です。一応 C によるエミュレーションで全数チェックを行って、x87 の fist 命令の結果と完全に一致することを確かめました。

4.5 FLOOR

4.5.1 実装基準と解釈

有効な浮動小数点数 A が $-2^{127} < A < 2^{127}$ を満たすとき、FLOOR(A) は有効な浮動小数点数を返し、その値は整数で、かつ、FLOOR(A) $\leq A <$ FLOOR(A) $+ 1$ が成立すること。

絶対値が 8388608.0 以上の浮動小数点数は、元々整数値なので何もする必要がありません。よって、$|A| < 8388608.0$ の場合のみ考えればよいことになります。

4.5.2 実装概要

$B =$ ITOF(FTOI(A)) を求めればだいたい正しい値が得られそうです。ただし、FTOI が round-to-even の丸めを行うので、$B - 1.0 < A < B$ となることがあります。このような場合は、1.0 を FSUB して補正します。$|A| < 8388608.0$ なので、この FSUB は exact に行えます。

また、FTOI の「0x4b000000 を引く sub」と ITOF の「0x4b000000 を足す add」は相殺することができます。よって、FADD($|A|, 8388608.0$) → FSUB($|A|, 8388608.0$) → 符号部を調整 → 1.0 を引くかどうか判定、としてやればよいことになります。

4.5.3 誤差評価

FTOI の丸め処理が正しければ大丈夫です。一応 C によるエミュレーションで全数チェックを行って、x87 の frndint 命令 (丸めモードを切り下げに変更) の結果と完全に一致することを確かめました。

5 既知のバグ

5.1 FDIV

$2^{126} \leq A < 2^{127}$ のとき、A の指数部が 0xfd なので、指数部計算の仕様上 FINV(A) はゼロを返します。FDIV(A, A) を考えると、$-2^{127} < A, A/A < 2^{127}$ であるにもかかわらず、FDIV(A, A) はゼロになってしまいます。

対策としては、FDIV が呼ばれるたびに (ソフトウェアで) 指数部のチェックを行い、指数部が 0xfd だった場合は別に処理する、という方法が考えられます。しかし、このようなコーナーケースに対応するために毎度分岐命令が入るのは煩わしいので、特に対策はせず、ここにバグとして記しておくにとどめます。

6 まとめ

FADD, FSUB, FMUL, FDIV(FINV), FSQRT, SIN, COS, ATAN, ITOF, FTOI, FLOOR の各浮動小数点演算について、実装の詳細を示し、誤差評価を行いました。FDIV のバグを除けば、浮動小数点演算実装基準を満たしていることを示せたのではないかと思います。

謝辞

CPU 実験という存分に遊べる環境を提供してくださった菅原さん、TA の伊藤さんに感謝いたします。

CPU 実験に関する数々の資料を残してくださった先輩方に感謝いたします。特に、FPU の設計にあたっては、2005 年度の平野班、紅魔館のコードを大いに参考にさせていただきました。

FPU の実装について議論をしてくださった 2006 年度 5 班 (cpu-5) の花岡君、2007 年度 1 班の小泉君に感謝いたします。

三角関数の精度の問題を指摘してくださった 2007 年度 1 班の植村君に感謝いたします。彼の指摘がなければ、三角関数の実装を見直すことはなかったでしょう。

また、実験が終わった後もタイムを縮め続ける私を生温かい目で見守ってくださる Team Sakanaya の皆様に感謝いたします。

参考文献

[1] 浮動小数点演算プリミティブの実装基準 (仮).

[2] 小林芳直. ディジタル・ハードウェア設計の基礎と実践—高性能、高信頼性システムを開発するための定石. CQ 出版社, May 2006.

[3] Robin Green. Faster math functions. Game Developers Conference 2003, http://www.research.scea.com/gdc2003/fast-math-functions.html, March 2003.

[4] 来犬リュウジ. Sin,Cos を頑張ってみる. http://www22.big.or.jp/~qul/program/text/route003.html.

更新履歴

2007/12/01
書き始め。FADD について書いている途中で基準を誤解していたことに気づいてショックのあまり中断。

2008/01/16〜18
復活〜とりあえず完成。

2008/03/08〜09
FINV の数式修正。(Thanks to 植村君)
FLOOR の説明が不適切だったのを若干修正。
その他細かい表現を修正。

2008/12/16
三角関数の評価が不適切だったのを若干修正。
その他細かい表現を修正。