# URR浮動小数点数のための高速演算装置の基本設計と実装

大山　光男
倉敷芸術科学大学産業科学技術学部

（2007 年 10 月 10 日　受理）

## １．はじめに

コンピュータ内部における数値表現として，科学技術計算では，浮動小数点表現が使われる。現在では，1985 年に標準化されたIEEE 754 標準が多く使用されているが，表現範囲，表現精度の点で充分でないアプリケーションもあり，可変長の指数部を持つ表現がいくつか提案されている[(1)]。なかでも浜田の提案したURR（Universal Representation of Real numbers）は[(2)][(3)]，エレガントな表現に加え，事実上，オーバーフローもアンダーフローも発生しない，データ長に独立である，などの好ましい特徴を持つことから注目された。一方，その実用化にあたっては，演算速度での不利，精度分析の難しさが指摘された。精度分析に関しては，事例ではIEEE との比較で改善される場合が多いことが報告されているが[(4)]，理論的な分析は充分行われていないようである。

筆者らは，演算速度での不利を克服するべく，高速演算器のアーキテクチャと，その実証システムを開発してきた[(5)]。仮数と指数の分離・結合を，それぞれパイプラインの１ステージで行うことを可能とする高速回路の開発[(6)-(8)]，さらに，分離・結合ステージを演算パイプラインから削除することにより，IEEE 標準を演算するFPU（Floating-Point Unit）のスループットに迫る性能を得る見通しが得られつつある[(9)-(13)]。本稿では，その実証システムである，64 ビットFPU の設計と実装，および評価結果について述べる。

## ２．URR浮動小数点数の概要

URR の定義は実数値の区間分割に基づくが[(2)]，浮動小数点数としては，IEEE 標準が固定長の指数部を持つ（図１）のに対して，図２に示すように指数のサイズに応じて長さが変わる可変長の指数部を持つ表現として解釈される。各部の表現は以下のようになる。

実数 x を，$x = f \times 2^e$ と表す。e は指数，f は仮数，ともに負数は２の補数で表し，正規化条件は，$f < 0$ では $-2 \leqq f < -1$，$f \geqq 0$ では $1 \leqq f < 2$ である。$f \geqq 0$，$e \geqq 0$ で，e が m 桁で $01e_{m-3}\cdots\cdots e_0$ と表されるとき，m ビットの１のビット列に反転ビット０をデリミッタとして連結，L 部とし，$e_{m-3}\cdots\cdots e_0$ を連結してE 部とする。すなわち，$11\cdots\cdots 10e_{m-3}\cdots\cdots e_0$ が指数部となり，仮数の符号を符号ビットS，仮数の小数点以下のビット列を指数部に連結して仮数部とする。$e < 0$ の場合は，L 部のビット列を 1/0 反転する。ま

た，e が −2，−1，0，1 の各場合は E は空であり，L は，それぞれ，001，01，10，100 である。さらに，x ＜ 0 の場合は，上記 L，E の各ビットを 1/0 反転して指数部とする。

図 4 に示すように，URR の指数部は，指数のサイズが小さい範囲では IEEE 標準に比べても指数部の長さは短いが，指数のサイズが大きくなると指数部の長さが急激に増す。これを改善する表現が提案されているが[14][15]，ここでは，拡張 URR を演算の対象に加える。拡張 URR は，2 重指数分割を URR が $\pm 2^{\pm 2^m}$ で行うのに対して，$\pm p^{\pm q^m}$ で行う。浮動小数点数としての解釈は図 3 に示すようになり，図 4 にも示すように，指数のサイズが大きい領域で，指数部の長さが大幅に短くなる（p = 4，q = 16 の場合）[15]。

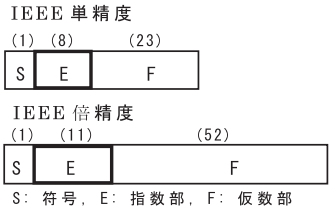



図1．IEEE データフォーマット

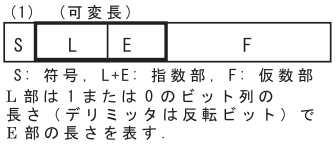



図2．URR データフォーマット

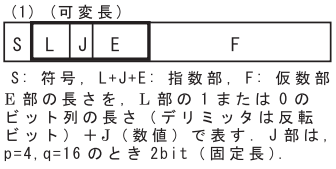



図3．拡張 URR データフォーマット

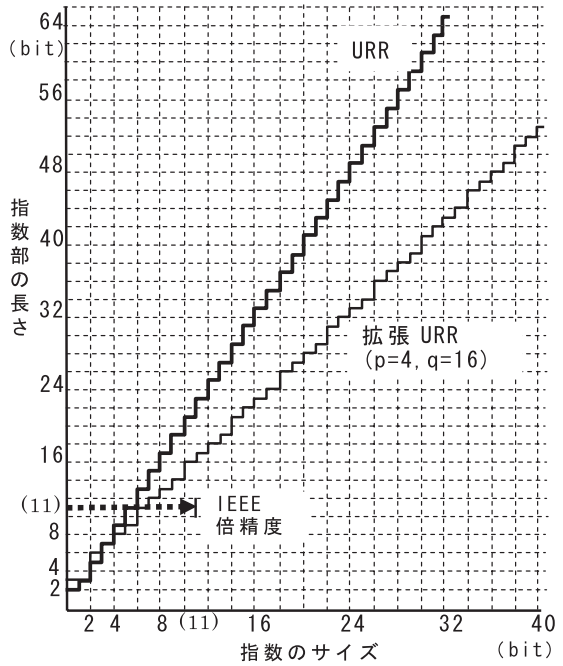



図４．指数のサイズと指数部の長さ

## 3．高速演算方式

演算は，指数と仮数を分離して行うので，指数と仮数の分離・結合の高速化が最初に課題となった。次に，演算パイプラインの指数と仮数の分離・結合の各 1 ステージを削除して，演算パイプのレイテンシを IEEE 標準と同等まで短縮することを目標とした。

### 3.1　指数と仮数の分離・結合の高速化

分離では，指数部 L のデリミッタビットの位置を検出することにより各部の境界を決定する。L 部の左端のビットを基準に右に調べ，最初の反転ビットの位置が L 部の右端である。類似の技術は，正規化回路の LZA（Leading Zero Anticipator）で使用される。ここでは，プライオリティエンコーダを基本構成要素とし，ビット並列に検出する。図 5 に示す分離回路では境界検出回路がそれである。境界検出回路の出力からビットパターン発

生器を用いてＬ部を指数の符号ビットを変換，さらに左右のバレルシフタを制御して指数と仮数に分離する。結合は図６に示す回路により行う。境界検出回路に類似した指数桁数検出回路で指数のサイズを検出，指数を指数部の位置にシフトした後，ビットパターン発生器を用いて指数部を生成する。同時に，仮数をシフトして，指数部に連結する。これらの回路は，64 ビット URR の場合，回路規模は 3500 ～4000 ゲート程度であり，パイプラインの１ステージで充分実行できることが示されている[6][7]。

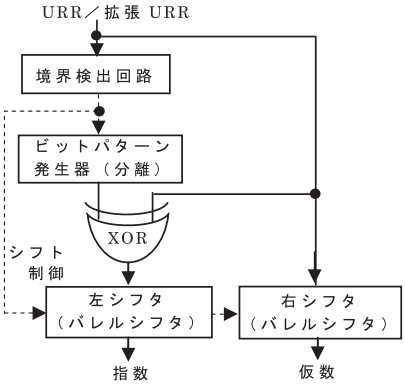



図５．分離回路の構成

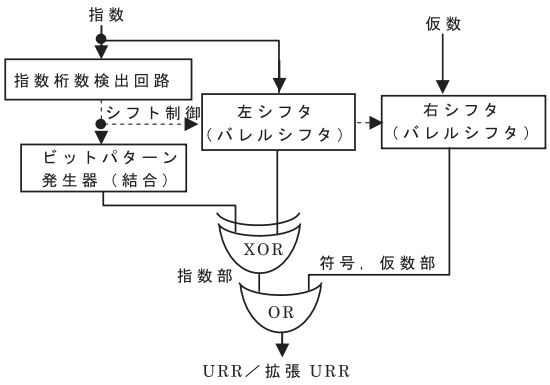



図６．結合回路の構成

### 3.2　演算パイプラインのレイテンシ短縮

レイテンシの短縮は，命令実行のスケジューリングを容易にするので，スループットの低下を抑える上で効果が大きい。そこで，指数と仮数を分離して浮動小数点レジスタに格納，分離・結合の２ステージを演算パイプから除去する。分離はロード命令に続いて独立した１ステージで，結合は，演算命令に続いて独立した１ステージで実行，可能な限り他の処理にオーバラップ，隠蔽することにより，パイプのストールの発生を抑制する。

## ４．FPUの設計

実証評価マシンとして，64 ビット URR，拡張 URR を対象とする FPU を設計，実装した。

### 4.1　仕様

表１に示すように，64 ビット URR から指数 32 ビット，仮数 64 ビットを分離して浮動小数点レジスタに格納する。指数 32 ビットは，表現からの制約ではなく，本 FPU ハードウェア独自の制約である。バッファレジスタには，浮動小数点レジスタと同一内容を，ストア命令

表１．FPU の仕様

| 項　目 | 仕　様 |
|---|---|
| データ | URR(64bit) / 拡張 URR(64bit) ⇄（分離／結合）⇄ 指数(32bit) / 仮数(64bit) |
| 浮動小数点レジスタ | 浮動小数点レジスタ：(32bit+64bit)×32 本<br>同バッファレジスタ：64bit×32 本 |
| 命令 | 浮動小数点：加算，減算，乗算，除算，積和演算，平方根，ロード／ストア，レジスタコピー，他．<br>整数：フロー制御命令，他． |
| モニタ | クロック数カウント<br>指数サイズの出現度数カウント |

に先行して URR の形で格納しておく。

評価用のモニタとして，プログラムの実行に要するクロック数のカウント，指数サイズの出現度数のカウント機能を設ける。

### 4.2　命令セット

いくつかの実用的なプログラムの実行を可能とするため，除算，平方根演算を含む主要な浮動小数点命令，およびフロー制御のための整数命令を実装する。命令語は 32 ビットの単一命令語長である。実装した命令の詳細は付録として添付した。

### 4.3　FPUの構成

図７に FPU の構成を示す。分離，結合ユニットは演算パイプには含まれない。ロード命令はデータメモリから URR を読み出してバッファレジスタに書き込むと同時に分離回路にセット，分離回路は独立した１ステージで指数と仮数を分離，浮動小数点レジスタに書き込む。結合は，ストア命令に先行して，演算命令により浮動小数点レジスタの内容が更新されるごとに独立した１ステージで行い，バッファレジスタに書き込む。バッファレジスタは浮動小数点レジスタのコピーを URR で持ち，ストア命令のソースオペランドとなる。

### 4.4　命令実行のレイテンシとスループット

表２に示すように，非パイプライン処理の除算と平方根以外のスループットは１クロックである。除算と平方根は，高速乗算器を利用し，Newton-Raphson 法をベースとする漸

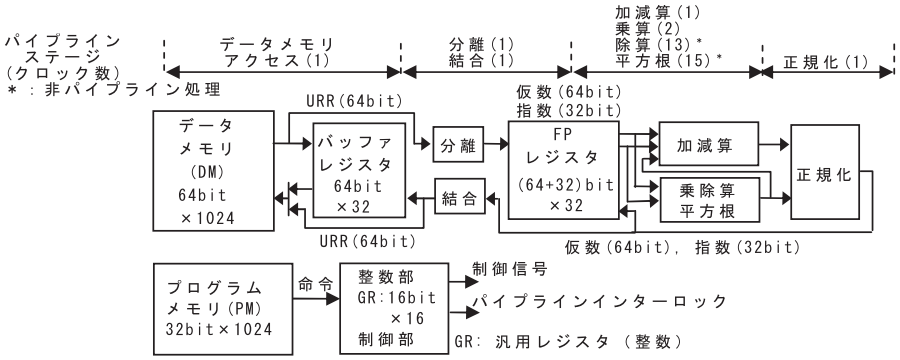



図７．FPU の構成とパイプラインステージ

表２．命令実行のレイテンシとスループット

| 命　令 | レイテンシ | スループット |
|---|---|---|
| ロード／ストア | 1 ／1 | 1 |
| 加算，減算，絶対値，レジスタコピー | 2 | 1 |
| 乗算／積和 | 3 ／ 4 | 1 |
| 除算／平方根 | 15 ／17 | 13 ／15 |
| 整数命令 | 1 | 1 |
| （分離／結合） | 1 ／1 | 1 ／1 |

註：スループット：同一命令を次に受け付けるまでに必要な時間（クロック）．　単位：クロック

近演算を実装する[16]。平方根の演算では，初期値テーブルのルックアップで高速化を図っている。繰り返し回数は $\log_2 n$ オーダーであり，非パイプライン処理ではあるが，除算で 13 クロック，平方根で 15 クロックのスループットを得ており，高速である。

## 5．実装

全体を 2M ゲートクラスの FPGA（XC3S2000）1 個に実装[17]，開発ツールは ISE Foundation 7.1i を使用した[18]。乗算を高速化するため，FPGA がハードマクロで内蔵する乗算器（18 × 18bit）11 個を用いて，符号付き 64 × 64bit の 2 段パイプライン並列乗算器を構成した。これにより，除算，平方根の演算の高速化も可能になった。データメモリ，プログラムメモリ，および浮動小数点レジスタの実装には，やはりハードマクロで内蔵される，2 ポート 16kb ブロック RAM15 個を利用した。アクセスが集中する浮動小数点レジスタは，書き込み 2 ポート，書き込み 1 ポート＋読み出し 2 ポート，および読み出し 4 ポートの 3 つのモードで動作可能なように構成している。全体の実装には，ハードマクロを除き，URR 対応では LUT（4 入力 Look Up Table）が 6825 個，拡張 URR 対応では 6905 個が使用された。

## 6．評価

分離・結合ステージの隠蔽の観点からスループットを，URR と拡張 URR との比較で演算結果に含まれる相対誤差を評価，拡張 URR の効果を検証する。

### 6.1　分離・結合ステージの隠蔽

（1）分離ステージ

分離に起因するストールが発生するとスループット低下の要因となり得る。ストールが発生するのは，図 8 に示すように，先行のロード命令の結果（R1 の内容）を後続の命令が直ちに利用する場合である。この場合，後続 2 命令の実行順序を入れ替えることにより容易に回避できる。入れ替えができない場合にのみ IEEE 標準に比べて 1 クロックの不利が生じる。

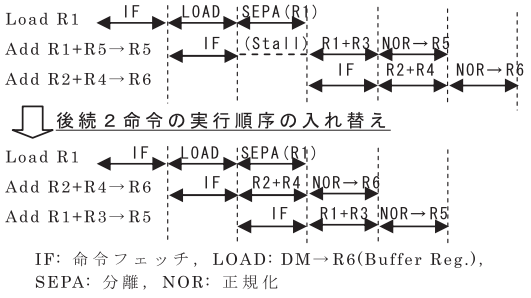



**図 8．分離に起因するストールの発生と回避例**

（2）結合ステージ

結合に起因するストールが発生するのは，先行命令が更新した浮動小数点レジスタの内容を，後続のストア命令が直ちに格納の対象とする場合である。図 9 に示すように，本 FPU の場合，最大 2 クロックのストールが発生し，このうち，結合に起因するのは 1 クロッ

クであり，他の１クロックはIEEE標準でも共通に発生する。回避は，後続命令の実行順序を入れ替えるスケジューリングにより行うが，スケジューリングができない場合のみ，結合に起因するストールが発生し得る。

これらのストールは，多くの場合，コンパイラによる静的なスケジューリングで対応が可能であり，IEEE標準に比べての不利はごくわずかに抑えられると考えられる。

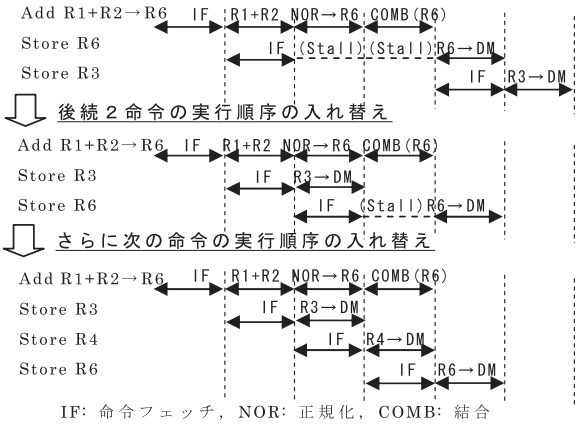



図９．結合に起因するストールの発生と回避例

### 6.2　事例における評価

（１）ワークロード

Graeffe法による４次方程：$a_4X^4+a_3X^3+a_2X^2+a_1X+a_0 = 0$の根の計算プログラムを実装した。図10に処理のフローを示す。プログラムは，根の値を絶対値で得るが，符号までは求めていない。マシン命令によるプログラムリストは，付録として添付した。Graeffe法では，係数の更新で乗算を繰り返すので，急速に係数の値が大きくなり，IEEE標準の浮動小数点表現ではオーバーフローが頻発，充分な繰り返し演算ができない。

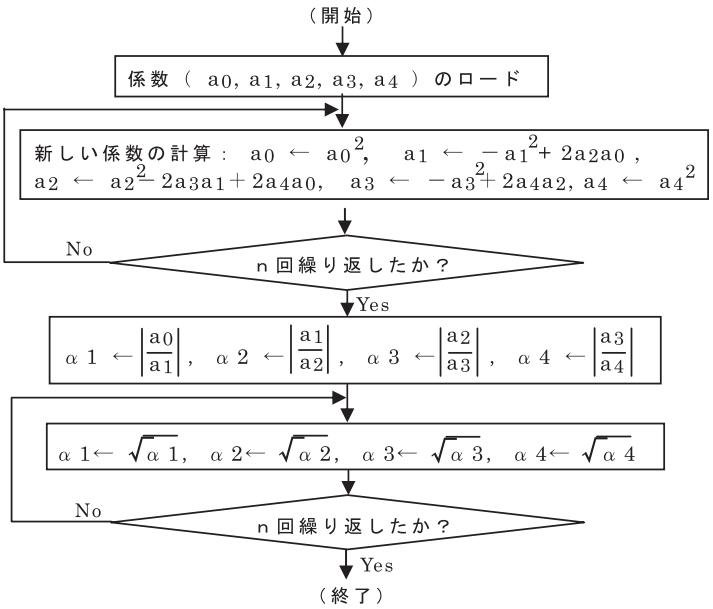



図10．プログラムのフロー（Graeffe法，４次方程式）

（2）スループット

分離・結合に起因するストールの発生がなければ，分離・結合がスループット低下の直接の要因にはならないと考える。そこで，プログラムが実行する各マシン命令の数と，そのマシン命令のスループットの積の総和から，プログラムの実行に要するクロック数を大まかに見積もり（表3），実測値と比較した。プログラムは付録に表示したプログラムで，繰り返し回数を1回に設定した場合である。6個の分離ステージ，29個の結合ステージは全て隠蔽されるものとして見積もりには含めない。FPUでのクロック数のカウントは146であり，ほぼ見積もりに一致する。1クロックの増加は正規化ユニットの競合からであり，分離・結合に起因しないことが命令実行シーケンスの分析から確認されている。

なお，表3の見積もりに使用したプログラムでは，ストア命令が含まれていない。そこで，後述のロード命令とストア命令の実行を多数含む改変プログラムを用いてクロック数の見積もりと，実測を行った。その結果，見積もりと実測のクロック数が一致し，実行シーケンスの分析からも，分離・結合ステージがすべて隠蔽されることを確認している。

表3．プログラム（繰り返し1回）の実行に要するクロック数の大まかな見積もり

| 命令 | 実行数 | スループット | クロック数 | 分離 | 結合 |
|---|---|---|---|---|---|
| 最初の命令フェッチ | | | 1 | | |
| ロード | 6 | 1 | 6 | 6 | |
| 加算，絶対値 | 8 | 1 | 8 | | 8 |
| 乗算 | 13 | 1 | 13 | | 13 |
| 除算 | 4 | 13 | 52 | | 4 |
| 平方根 | 4 | 15 | 60 | | 4 |
| 整数命令 | 5 | 1 | 5 | | |
| （合計） | 40 | | 145 | （6） | （29） |

（3）演算結果に含まれる相対誤差

4次方程式（x−2）（x−e）（x−$\sqrt{7.4}$）（x−3）=0の根をGraeffe法で計算し，計算した根に含まれる相対誤差を評価した。このプログラムでは，計算過程ではロード／ストアが行われない。すなわち，結果に影響する分離･結合は行われないので，図11に示すように，URRか拡張URRかは関係なく，収束後は繰り返しを重ねても誤差はフラットである。なお，値が近接する，eと$\sqrt{7.4}$の収束が良くないのはGaraeffe法自体に起因する。IEEE倍長の場合は，オーバーフローの発生により，この場合，繰り返しは8回が限界である。

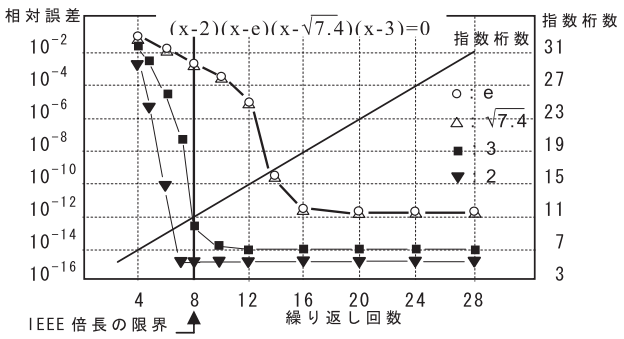



図11．演算結果の相対誤差（Graeffe法，URR／拡張URR）
（計算過程で分離、結合が行われない場合）

ちなみに，IEEE 倍長の表現範囲はおおよそ $10^{-308}$ ～$10^{308}$ であるのに対して，URR（指数 32 ビットの場合）ではおおよそ $10^{-646392578}$ ～$10^{646392578}$ である。

次に，指数のサイズが大きい領域での拡張 URR の効果を評価するため，付録に表示したプログラムを改変し，演算命令が浮動小数点レジスタを更新するごとに，更新したレジスタの内容を，一度データメモリに書き戻し，再びロードし直すようにした。これは，演算精度の点からは，演算パイプに分離結合ステージを含む場合に相当し，分離・結合ステージの隠蔽の点からは，ロードストア命令を頻繁に実行する，厳しい環境となる。

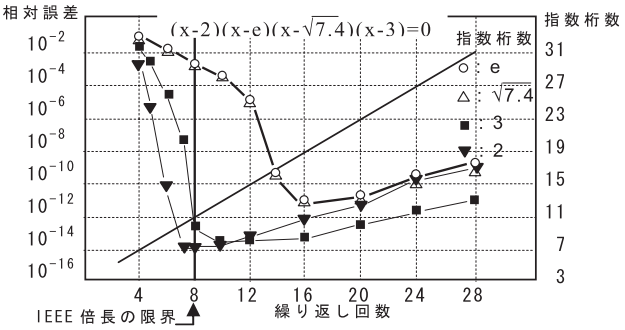



図 12. 演算結果の相対誤差（Graeffe 法，URR）
（演算ごとに分離結合を繰り返す場合）

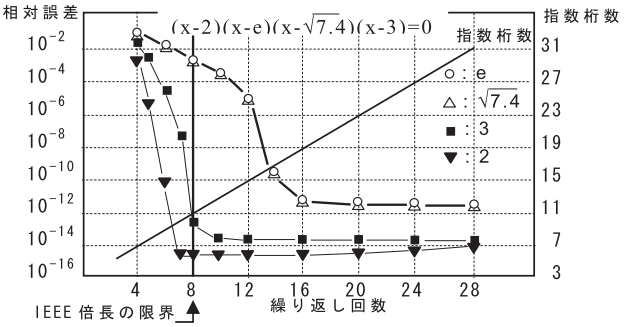



図 13. 演算結果の相対誤差（Graeffe 法，拡張 URR）
（演算ごとに分離結合を繰り返す場合）

図 12 に示すように，URR では，一度収束した後，さらに演算を繰り返すと，誤差が漸増する。これは，データ長が一定の場合（64 ビット），指数サイズが大きい領域では仮数部のビット数が充分確保できなくなるからであろう。URR の場合，指数のサイズが 32 ビットで仮数部のビットが確保できなくなる。一方，図 13 に示すように，拡張 URR では，指数のサイズが 32 ビットに近づいても誤差の増加はわずかである。実際，指数が 32 ビットの場合も仮数部は 22 ビットを確保している。

## 7．考察

演算パイプから分離・結合ステージを削除したことにより，演算パイプのレイテンシの不利が完全に解消された。演算パイプの外で行う分離・結合に起因するパイプのストールの発生は限定的であり，発生する場合でも，多くは静的な命令実行のスケジューリングによる回避が期待できる。事実，本 FPU での事例では，スループット低下の要因になっていない。しかし，FPU の設計パラメータは広範にわたり，かつ動作環境も多様である。より信頼性のある評価を得るには，さらに完成度を上げた FPU を実現，多様な動作環境下，

多くのアプリケーションプログラムでの評価を積むことが必要である。

また，指数と仮数は分離して浮動小数点レジスタに格納されるので，演算命令の演算精度が指数のサイズの影響を受けず，常に最大の精度が得られる。ただ，浮動小数点レジスタの数は限られるので，途中結果をメモリに一時格納する場合，指数のサイズが極端に大きいと，結果の精度に影響を及ぼす可能性があるが，拡張 URR では影響が大幅に軽減されるし，分離したままロード／ストアできるよう，命令を拡張するなどの対応も考えられる。

本 FPU では，分離・結合回路以降の，浮動小数点レジスタを含む演算器の構成は，データフォーマットに依存しない。内部はデータフォーマットに独立で，外部とのインターフェースを，URR のようにデータ長独立の表現方式で統一した，新しい FPU の実装方式が可能ではないだろうか。

## 8．おわりに

URR 浮動小数点数のための高速 FPU の開発を行ってきたが，IEEE 標準の浮動小数点数を演算する FPU のスループットに迫れる可能性は見いだせたと考えたい。また，考察でも述べたように，内部演算における数値の範囲や精度を，データフォーマットに依存せず柔軟に設定でき，かつ統一した外部インターフェースを持つ FPU 実現の可能性を探りたい。

### 参考文献


(1) 松井正一，伊理正夫：あふれのない浮動小数点表示方式，情報処理学会論文誌，Vol.21, No.4, pp.306-313,（1980）.

(2) 浜田穂積：２重指数分割に基づくデータ長独立実数値表現法Ⅱ，情報処理学会論文誌，Vol.24, No.2, pp.149-156,（1983）.

(3) Hamada, H. : A New Number Representation and Its Operation,Proc.of 8th Symposium on Computer Arithmetic, pp.153-157, Como, Italy,（1987）.

(4) 菊池純男，他：URRアーキテクチャおよびコンパイラの試作，情報処理学会第37回全国大会，5D-10,（1988）.

(5) 大山光男，浜田穂積：URR浮動小数点数演算のための指数仮数高速分離・結合回路方式とURRプロセッサへの応用，情報処理学会論文誌，Vol.35, No.8, pp.1642-1651,（1994）.

(6) 大山光男：３重指数分割に基づく浮動小数点数演算のための指数と仮数の高速分離結合回路の設計と評価，情報処理学会論文誌，Vol.37, No.4, pp.613-623,（1996）.

(7) 大山光男：URR浮動小数点数演算のためのパイプライン加減算器の設計とFPGAによる実現，情報処理学会研究報告，97-HPC-64-4, pp.19-24,（1997）.

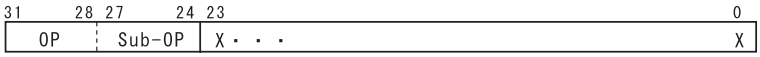
(8) Mitsuo Ooyama : Fast Separation and Combination of an Exponent and a Fraction for URR Floating-Point Arithmetic and Its Application to a Pipelined Adder/Subtracter，GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics（SCAN97）, Lyon, France,（1997）.

(9) Mitsuo Ooyama : A Design of a fast Floating-Point Unit for URR Floating-Point Arithmetic, GAMM-

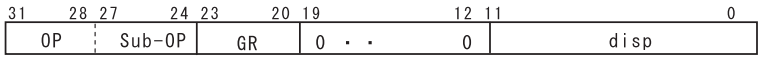
IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics（SCAN2000）, pp.96-97, Karlsruhe, Germany,（2000）.
（10）Mitsuo Ooyama : A Floating-Point Unit with Reduced Latency for URR Floating-Pont Arithmetic, GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics（SCAN2004）, p.92, Fukuoka, Japan,（2004）.
（11）大山光男：レイテンシを短縮した32ビットURR浮動小数点数演算器の設計と実装，第4回情報科学技術フォーラム（FIT2005），B-028,pp.157-158,（2005）.
（12）Mitsuo Ooyama : A 64-bit High Performance Floating-Point Unit for URR and Extended URR Floating-Point Arithmetic, GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics（SCAN2006）, pp.49-50, Duisburg, Germany,（2006）.
（13）大山光男：URR浮動小数点数のためのレイテンシを短縮した64ビットFPUの実装，第６回情報科学技術フォーラム（FIT2007）, B-001, pp.77-78（2007）.
（14）中森真理雄，土井　孝：３重指数分割による数値表現方式について，電子情報通信学会論文誌，Vol. J72-A, No.7, pp.1468-1469,（1988）.
（15）富松　剛：拡張した二重指数分割による数値表現法に関する研究，情報処理学会研究報告，95-HPC-58, pp.57-62,（1995）.
（16）Behrooz P. : Computer Arithmetic, Oxford University Press, New York,（2000）.
（17）http://japan.xilinx.com/products/silicon_solutions/（2007）.
（18）http://japan.xilinx.com/ise/logic_design_prod/foundation.htm（2007）.

## 《付録１：実装したマシン命令の一覧》

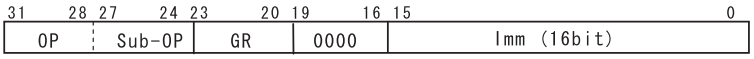
### １．整数命令




0000 0000 NOP (No Operation)
0000 0001 HLT (Halt)



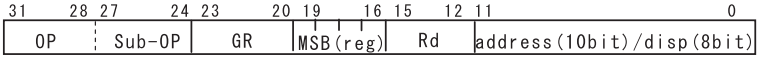

0001 0000 BRA (Branch Always):PC ← PC+1+disp, －2048≦disp≦2047
0001 0001 BEZ (Count and Branch on Equal Zero):
GR ← GR－1,if GR=0,then PC ← PC+1+disp,otherwise PC←PC+1
0001 00 0 BNZ (Count and Branch on Not Equal Zero)
GR ← GR－1,if GR≠0,then PC ← PC+1+disp,otherwise PC←PC+1




0010 0000 LDI (Load Integer Immediate): GR ← Imm
0010 0001 ADDI (Add Integer Immediate): GR ← GR + Imm
0010 0010 SUBI (Subtract Integer Immediate): GR ← GR－Imm

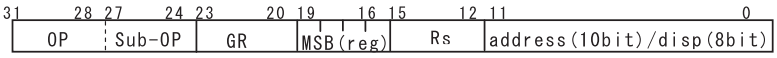
### ２．浮動小数点命令

（Load 命令）



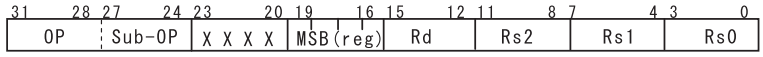

1000 xxxx FLD (FP Load Direct); Rd ← Separate(DM(address))
1001 0000 FLRI (FP Load Register Indirect): Rd ← Separate(DM((GR)+disp)))
1001 0010 FLRI (FP Load Register Indirect, Pre_increment):
GR ← GR+1, Rd ← Separate(DM((GR)+disp)))
1001 0011 FLRI (FP Load Register Indirect , Post_decrement):
Rd ← Separate(DM((GR)+disp))),GR ← GR－1

（Store 命令）




1010 xxxx FSD (FP Store Direct ):DM (address) ← Combine(Rs)
1011 0000 FSRI (FP Store Register Indirect ): DM((GR)+disp) ← Combine(Rs)
1011 0010 FSRI (FP Store Register Indirect, Pre_increment):
GR ← GR+1,DM ((GR)+disp) ← Combine(Rs)
1010 0011 FSRI (FP Store Register Indirect, Post_decrement):
DM((GR)+disp) ← Combine(Rs), GR ← GR－1

（演算命令）




1100 0000 FMOV ( FP Move ) : Rd ← Rs0
0001 FNMOV ( FP Negative Move ) : Rd ← －Rs0
0010 FABS ( FP Absolute ) : Rd ← | Rs0 |
0011 FCMP ( FP Compare ) : Rs1 － Rs0, Set Status.
0100 FADD ( FP Add ) : Rd ← Rs1 + Rs0
0101 FSUB ( FP Subtract ) : Rd ← Rs1 － Rs0
0110 FMLT ( FP Multiply ) : Rd ← Rs1×Rs0
0111 FDIV ( FP Divide ) : Rd ← Rs1÷Rs0
1000 FMADD ( FP Multiply and Add ) : Rd ← Rs1×Rs0＋Rs2
1001 FMSUB ( FP Multiply and Subtract ) : Rd ← Rs1×Rs0－Rs2
1010 FNMLT ( FP Negative Multiply ) : Rd ← (－Rs1)×Rs0
1011 FNDIV ( FP Negative Divide ) : Rd ← (－Rs1)÷Rs0
1100 1100 FSQRT ( FP Square Root ) Rd ← SQRT(Rs0)

註：記述のないオペレーションコード（OP, Sub-OP）については未定義であり，
実装していない．X は don’t care を表す．

## 《付録２．Graeffe 法による４次方程式の根の計算プログラム》

```
start:
        FLD R0,008          ;R0←2
        FLD R1,009          ;R1←a0
        FLD R2,00A          ;R2←a1
        FLD R3,00B          ;R3←a2
        FLD R4,00C          ;R4←a3
        FLD R5,00D          ;R5←a4
        LDI   GR1,n         ;ループ回数ｎ設定
loop1:
        FNMLT R7,R2,R2      ;R7←－a1
        FMLT  R8,R3,R3      ;R8←a2^2
        FNMLT R9,R4,R4      ;R9←－a3^2
        FMLT  R11,R3,R1     ;R11←a2a0
        FNMLT R12,R4,R2     ;R12←－a3a1
        FMLT  R13,R5,R3     ;R13←a4a2
        FMLT  R14,R5,R1     ;R14←a4a0
        FMLT  R11,R11,R0    ;R11←2a2a0
        FMLT  R12,R12,R0    ;R12←－2a3a1
        FMLT  R13,R13,R0    ;R13←2a4a2
        FMLT  R14,R14,R0    ;R14←2a4a0
        FMLT  R1,R1,R1      ;R1←a0^2              a0更新
        FMLT  R5,R5,R5      ;R5←a4^2              a4更新
        FADD  R3,R8,R12     ;R3←a2^2－2a3a1
        FADD  R2,R7,R11     ;R2←－a1^2+2a2a0       a1更新
        FADD  R4,R9,R13     ;R4←－a3^2+2a4a2       a3更新
        FADD  R3,R3,R14     ;R3←a2^2－2a3a1+2a4a0   a2更新
        BNZ   GR1,loop1     ;GR1←GR1-1,if GR1≠0 then go to loop1
;
        FDIV  R5,R4,R5      ;R5←a3/a4
        FDIV  R4,R3,R4      ;R4←a2/a3
        FDIV  R3,R2,R3      ;R3←a1/a2
        FDIV  R2,R1,R2      ;R2←a0/a1
        FABS  R5,R5         ;R5←α4, α4=|a3/a4|
        FABS  R4,R4         ;R4←α3, α3=|a2/a3|
        FABS  R3,R3         ;R3←α2, α2=|a1/a2|
        FABS  R2,R2         ;R2←α1, α1=|a0/a1|
        LDI   GR1,n         ;ループ回数ｎ設定
loop2:
        FSQRT R5,R5         ;R5←SQRT(α4)
        FSQRT R4,R4         ;R4←SQRT(α3)
        FSQRT R3,R3         ;R3←SQRT(α2)
        FSQRT R2,R2         ;R2←SQRT(α1)
        BNZ   GR1,loop2     ;GR1←GR1-1,if GR1≠0 then go to loop2
END:    HLT
```

註 1) ただし，定数 2，および４次方程式 $a_4X^4+a_3X^3+a_2X^2+a_1X+a_0=0$ の係数 a0,a1,a2,a3,a4 は DM(データメモリ)の 008 番地から順に格納されている．

註 2) 定数 2，および $(x-2)(x-e)(x-\sqrt{7.4})(x-3)=0$ の係数は，64bitURR，拡張 URR では以下のようになる．

```
64bit URR（16進表記）          64bit 拡張 URR(p=4,q=16,16進表記)
 2: 6000 0000 0000 0000        2: 5000 0000 0000 0000
a0: 7962 EFEF 9025 ECA5       a0: 6562 EFEF 9025 ECA5
a1: 85E9 956A 8E8C 6064       a1: 99E9 956A 8E8C 6064
a2: 7944 B301 BF3A E98F       a2: 6544 B301 BF3A E98F
a3: 8AC7 DCBE 7179 D716       a3: 9D63 EE5F 38BC EB8B
a4: 4000 0000 0000 0000       a4: 4000 0000 0000 0000
```

# Development of a High Performance Computing Architecture for URR Floating-Point Arithmetic and its Implementation

Mitsuo OOYAMA

*College of Science and Industorial Technology*
*Kurashiki University of Science and the Arts,*
*2640 Nishinoura, Tsurajima-cho, Kurashiki-shi, Okayama 712-8505, Japan*
(Received October 10, 2007)

This paper describes a novel FPU（Floating-Point Unit）for 64-bit URR and 64-bit extended URR floating-point arithmetic which achieves high performance close to that of the IEEE 754 FPU. URR has some desirable characteristics such as being free from overflows and underflows in practice, and its data format being independent of data length. However, it needs longer calculation process because of its complex exponent part.

The proposed architecture is based on an idea that reduces latency of the calculation pipeline. The FPU generates a 32-bit exponent and a 64-bit significand from a URR representation or an Extended URR representation, and store them to（32+64）-bit floating-point register respectively. The conversion to 96-bit format occurs immediately after the load instruction of the original data, and the reverse conversions are executed prior to the store instructions of the computed floating-point numbers. We implemented Graeffe's method to evaluate the FPU with practical programs. While the IEEE 754 FPU can not perform enough iterations necessary for Graeffe's method, because coefficients grow rapidly and exceed the maximum range of the exponent, but the proposed FPU can do. The total number of clock cycles necessary for the execution of the evaluation program on the proposed FPU is almost the same as that for the IEEE 754 FPU. We also evaluate the relative errors of the computed roots of the biquadratic equation to real values. While the computation can be carried out within the floating-point registers, relative errors are small and flat regardless of the size of the exponent in both URR and extended URR. On the other hand, when the floating-point calculations include load and store instructions for the intermediate values where the size of the exponent is very large, Extended URR achieves significant improvement in relative errors.