システムＬＳＩ設計技術　94－2
（2000. 1. 11）

# URRを用いた浮動小数点乗算回路のVLSIへの実装と評価

葛 毅　　阿部 公輝　　浜田 穂積

電気通信大学情報工学科

〒182-8585 東京都調布市調布ヵ丘 1-5-1

email: katsu-t@cacao.cs.uec.ac.jp, abe@cs.uec.ac.jp, hamada@cs.uec.ac.jp

本論文では，URR（Universal Representation of Real numbers）を用いた32ビット浮動小数点乗算回路のIEEE規格との比較とVLSIへの実装について述べる．URRとは浮動小数点数値表現法の一つである．URRは指数部と仮数部を可変長とすることで，IEEE規格に比べて遥かに大きな値や小さな値を表現することを可能としている．しかし，可変長であることから指数部と仮数部の分離/結合処理を行う回路を必要とする．本論文ではURRを実装する際の回路量を評価している．主に次について述べる．(1) URRを用いた浮動小数点乗算回路の構成と分離/結合を行う回路構成の詳細な検討．(2) 各構成要素の最適化．(3) IEEE規格の浮動小数点乗算回路との比較．IEEE規格との比較の結果，遅延時間で1.66倍，面積で2.52倍となった．なお，加算回路では遅延時間で1.68倍，面積で2.44倍となった．また，設計した乗算回路の試作チップを作成した．試作チップの主な製造条件は，CMOS 0.6$\mu$m，4.5mm角である．設計はVerilog-HDLで行い，論理合成にDesignCompiler（Synopsys社），配置配線にAquariusXO（Avanti社）を使用した．

キーワード：実数値表現，URR，IEEE754規格，浮動小数点乗算，VLSI

# VLSI Implementation and Performance Evaluation of URR Floating-Point Multiplier

Takeshi Katsu　　Kôki Abe　　Hozumi Hamada

Department of Computer Science, The University of Electro-Communications

1-5-1 Chofugaoka, Chofu-shi, Tokyo 182-8585 Japan

email: katsu-t@cacao.cs.uec.ac.jp, abe@cs.uec.ac.jp, hamada@cs.uec.ac.jp

In this paper we describe the design and VLSI implementation of a 32 bit floating-point multiplier where numbers are represented in an internal form named URR(Universal Representation of Real numbers) by the inventor. With exponential and mantissa parts of variable lengths, URR allows representation of much larger and smaller values than the IEEE standard. The variable length property, however, necessitates separation and combination of the exponential and mantissa parts. We investigate the cost of implementing URR by (1) designing a 32 bit multiplier with circuits for the separation and combination, (2) optimizing the components, and (3) comparing the results with IEEE standard implementation. The investigation reveals that the circuit complexity of URR multiplier is 1.66 times in delay and 2.52 times in area compared with that of IEEE multiplier. The costs of URR adder are also investigated in the same way, and found to be 1.68 and 2.44 for delay and area, respectively, taking IEEE adder's costs as the units. We realized the URR multiplier in a 4.5mm square VLSI chip with CMOS 0.6$\mu$m fabrication rule. The design tools used are Verilog-HDL for description, DesignCompiler for synthesis, and AquariusXO for placement and routing.

key words: real number representation, URR, IEEE standard 754, floating-point multiplier, VLSI

## 1 はじめに

IEEE754 規格の浮動小数点数値表現 [1] では，指数部と仮数部のビットが固定して割り当てられている．URR（Universal Representation of Real numbers）[2][3] と呼ばれる浮動小数点数値表現法では，指数部と仮数部を可変長として，指数部に多くのビットを割り当てることを可能にすることで，同じビット数では，IEEE 規格に比べて，遥かに小さな値や大きな値を表現することを可能としている．しかし，指数部と仮数部が可変長であることから，演算を行う為には，指数部と仮数部の分離や結合処理を行う回路が必要となり回路量が増える．

URR を用いた演算回路の回路量を評価するため，32 ビット浮動小数点乗算回路を設計した．浮動小数点数の演算では，乗算は仕組みが簡単であるため，設計が比較的容易であり，指数部と仮数部の分離/結合を行う回路の構成を詳細に検討するのに都合がよい．なお，加算の場合についても設計し評価を行なっている．性能評価は，論理合成ツールを用いて回路を作成し，その面積と遅延時間の数値を用いて，IEEE 規格との比較をすることで行った．合成に用いたセルライブラリは CMOS 0.6$\mu$m ルールである．ライブラリは VDEC（大規模集積システム設計教育研究センター）より提供されたものである．更に，設計した乗算回路のチップを試作した．設計は Verilog-HDL[4] で行い，論理合成に DesignCompiler（Synopsys 社），配置配線に AquariusXO（Avanti 社）を使用した．本論文では主に次について述べる．(1) URR を用いた浮動小数点乗算回路の詳細な構成と最適化．(2) IEEE 規格との比較．

本論文では，2 章で URR の概要，3 章で URR 乗算回路の基本構成，4 章で「符号なし」版 URR 乗算回路の構成，5 章で分離/結合回路の構成，6 章で「符号なし」版のモジュールの性能，7 章で「符号あり」版 URR 乗算回路の構成，8 章で IEEE 規格との比較，9 章で加算回路の比較，10 章で試作チップ，を述べる．

## 2 URR の概要

URR は指数部と仮数部の長さが可変であることが大きな特徴である．指数部と仮数部の境界は指数部の最上位ビットからの'0' または'1' の連続の個数を数えることで分かる仕組みになっている．ビット列は IEEE 規格と同様，「符号部」,「指数部」,「仮数部」の順である．符号部は 1 ビットで'0' のとき正，'1' のとき負である．実装において注意する性質としては，(1) IEEE 規格では指数部がゲタばき表現されているだけでそのまま演算できるのに対し，URR では指数部が特殊なビットパターンとなっているため，演算の可能な 2 の補数表現に変換しなければならないこと，さらに,「指数部」が同一のビットパターンであっても，値の正/負により表す「指数値」が異なり，変換方法が異なること，(2) IEEE 規格では仮数が正数であるのに対し，URR では 2 の補数表現を $1 \leq f < 2, -2 \leq f < -1$ の条件で正規化し，ケチ表現したものを仮数部としていること，(3) 符号の反転が符号ビットの反転ではなく，ビット列全体の 2 の補数をとること，がある．

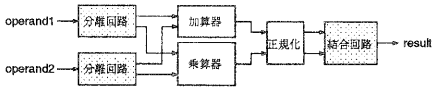


図 1: URR の乗算回路の基本構成

## 3 URR 乗算回路の基本構成

浮動小数点数の演算では通常，指数部と仮数部を分離して各々に対して演算を行うが，URR では境界が可変であることから，指数部と仮数部を分離する際,(1) 境界を見つける．(2) 指数部と仮数部を分離する．という処理を必要とし，また，指数部と仮数部の結合のため，同様の処理を演算後にも必要である．この分離/結合処理を行う部分を以下では「分離回路」「結合回路」と呼ぶ．後で述べる IEEE 規格との比較における分かりやすさのため，正規化をする部分は結合回路から外した．図 1に乗算回路の場合の概念図を示す．

また，IEEE 規格では仮数部は正数で表現されているため，仮数部の乗算は符号なし乗算アルゴリズムを用いている．一方，URR では，仮数部が 2 の補数で表現されているため，仮数部の乗算に符号なし/符号あり乗算のどちらを用いるかにより，分離/結合回路の構成が違ってくる．符号なし乗算アルゴリズムを用いた場合は，IEEE 規格に基づいた乗算回路を参考にして，分離/結合回路を付け加える形で URR 乗算回路全体を構成できるため，実装が比較的容易であり，分かりやすい．しかし，URR では仮数部が 2 の補数で表現されているため，仮数部を 2 の補数表現から正数に変換しなければならない．この場合，URR ではビット列の 2 の補数をとることで符号反転ができるという性質を利用して，値が負の場合は正の値にして乗算を行い，乗算結果が負となる場合には，乗算結果の 2 の補数をとるようにすれば，乗算回路自体は正数同士の乗算の場合のみを考えるだけでよい．一方，URR では仮数部が 2 の補数で表現されているため，仮数部の乗算に符号あり乗算アルゴリズムを用いれば，指数部と仮数部の分離をした際，分離された仮数部をそのまま使用できるため 2 の補数をとる必要が無く効率がよい．しかし構成が複雑になる．この各々を以下では「符号なし」版,「符号あり」版と呼ぶ．以下，まず分かりやすい「符号なし」版の構成を説明し，次に「符号あり」版を説明する．

## 4 「符号なし」版 URR 乗算回路の構成

「符号なし」版 32 ビット URR 乗算回路モジュール **urr_mul** のブロック図を図2に示す．実装した回路はこの「符号なし」版である．網目の部分が分離/結合回路である．それぞれモジュール **SEPARATE, COMBINE** とする．

乗算は，まず，モジュール **comp** でオペランドの 2 の補数を取っておく．次に，**mux** でオペランドの符号ビッ

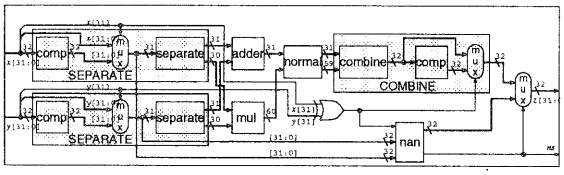


図 2: URR 乗算回路の構成

トにより値が正であるものを選ぶ．正の値となったものを **separate** で指数部と仮数部を分離し，指数値と仮数値に変換する．**adder** で指数値の加算を行い，**mul** で仮数値の乗算を行う．**normal** で指数値と仮数値の正規化を行い，**combine** で指数部，仮数部への変換と結合をして，IEEE 規格の nearest モードで丸めを行なう．それから分離回路の場合と同様に，結合した値の 2 の補数を **comp** で取っておき，演算結果が負の場合は 2 の補数を取ったものを選択し，出力とする．**nan** は非数の場合の表である．以上の流れと並列に，**nan** の表を引いて，非数の場合の結果を作っておき，結果が非数となる場合は，**nan** の結果を出力とし，$ns$ を 1 とする．分離/結合回路の本体である **separate** と **combine** の構成については次章で詳しく述べる．

## 5 分離/結合回路の構成

この章では **separate**, **combine** の構成について述べる．表 1に指数の変換の際に使用する「指数値」と「指数部」との対応表を示す．表中 $x_i \in \{0,1\}$ である．「指数部」とは URR の指数フィールドのことであり，「指数値」とは指数部の表す値を 2 の補数で解釈したもののことである．

### 5.1 モジュール separate の構成

モジュール **separate** は実際に分離を行う部分である．**separate** は，プライオリティエンコーダ（**prienc_s**）で境界を調べ，シフタ（**shifter_s**）でその値に基づいて指数部，仮数部を切り出して，残りのモジュールで指数部を対応する指数値に変換する．ブロック図を図 3に示す．

簡単のため，まず，クリティカルパスである仮数部の切り出しを行う，**prienc_s** から **shifter_s** のパスの構造から考える．まず，**prienc_s** で境界を調べる．境界は指数部

表 1: 「指数値」と「指数部」との対応表

| 指数値（31bit 2 の補数表現） | 指数部 | |
|---|---|---|
| | 値が正の場合 | 値が負の場合 |
| $1\cdots10\underbrace{x_{n-2}\cdots x_1}_{n-2}$ | $\underbrace{0\cdots01}_{n}\underbrace{x_{n-2}\cdots x_1}_{n-2}$ | $\underbrace{1\cdots10}_{n}\underbrace{\overline{x_{n-2}}\cdots\overline{x_1}}_{n-2}$ |
| $111\cdots110x_1$ | $0001x_1$ | $1110\overline{x_1}$ |
| $111\cdots1110$ | 001 | 110 |
| $111\cdots1111$ | 01 | 10 |
| $000\cdots0000$ | 10 | 01 |
| $000\cdots0001$ | 110 | 001 |
| $000\cdots001x_1$ | $1110x_1$ | $0001\overline{x_1}$ |
| $0\cdots01\underbrace{x_{n-2}\cdots x_1}_{n-2}$ | $\underbrace{1\cdots10}_{n}\underbrace{x_{n-2}\cdots x_1}_{n-2}$ | $\underbrace{0\cdots01}_{n}\underbrace{\overline{x_{n-2}}\cdots\overline{x_1}}_{n-2}$ |

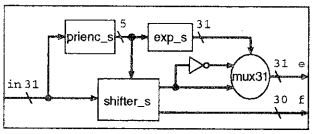


図 3: **separate** のブロック図

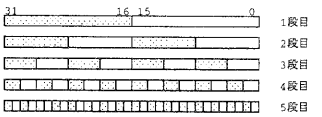


図 4: **prienc_s** の構造の概念図

の先頭からの'0' または'1' の連続の数から分かる．指数部の先頭が'0' の場合は，ビット列全体を反転させておき，'1' の連続の場合についてのみ調べる．ビット長を $N$ とすると **prienc_s** では，'1' の連続を半分づつ見ていくことにより，$\log_2 N$ 段で連続の個数を 5 ビットにエンコードする．概念図を図 4に示す．図中の大きな長方形は符号部を除いて，最下位ビットに 0 を付け加えたビット列である．長方形の網目部が全て'1' であるかどうかを見ていき，'1' の連続の個数を 5 ビットにエンコードする．

**shifter_s** では，**prienc_s** でエンコードされた値に基づいて，入力されたビット列をバレルシフタを用いて左シフトすることにより，仮数部を取り出す．バレルシフタは 5 段である．プライオリティエンコーダで 5 ビットにエンコードされた値は'1' の連続の数である．この数を $n$[4:0] とすると，指数部は $2n-1$ ビットであるから（表 1），$2n-1$ ビット左シフトすることで，仮数部を取り出すことができる．シフトでは，$n[i]$ が立っている場合は $2^i$ を表しているから，$2^i \times 2 = 2^{i+1}$ ビットシフトすることで，$2n$ シフトすることができる．ただし，ここでシフトしたい数は $2n-1$ であるから，シフトするどこか一か所で定数 $-1$ を加えておけばよい．指数部は少なくとも 2 ビット以上あり，$n$ は 0 となることはないので，$n$ の各ビットの少なくとも一つは 1 が立っているから，$-1$ ビットシフト（1 ビット右シフト）だけがされることはない．

ここで，**prienc_s** では，$n$[4] から $n$[0] の順に，上位ビットから先に決まるので，**shifter_s** では，$n$ の上位ビット，つまり $n$[4] から先に使用してシフトしていくことで，遅延をオーバーラップさせることができる．ブロック図を図 5に示す．図では $n$[0] によるシフトの際に $-1$ を加えて，$2n$[0] $-1$ ビットシフトとしている．また，$n$ が 1 のときの指数部のビット長は $2n-1=1$ ではなく 2 であるため，この場合は別に作っておき，最後に選択する．以上の構成から分離回路は時間的に $O(\log N)$ である．

次に，指数値の取り出しについて述べる．指数値の取り出しは，まず，指数部を取り出して，次に取り出した指数部を対応する指数値（2 の補数表現）に変換することで行う．指数部の取り出しは，先の仮数部の取り出しと同じ

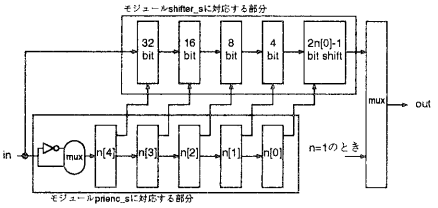


図 5: 仮数部の取り出しの回路構成

である．つまり，シフタ（**shifter_s**）のビット長を倍に拡張することで，仮数部と同時に指数部を取り出すことができる．この場合，指数部の取り出しにおけるシフトの際に，符号部を除いた指数部の最上位ビットを拡張する必要がある．次に取り出した指数部を対応する指数値に変換する．今は値が正の場合しかないため，表 1の正の場合だけを考える．表 1の実装は，**prienc_s** の出力 $n$ に基づいて，下位から数えて $n-1$ ビット目以上のビットは反転し，$n-2$ ビット目より下位のビットはそのままにすることで行う．具体的には，まず，取り出された指数部の各ビットを反転させたものを作っておき，マルチプレクサで「反転させたもの」と「そのままのもの」を選択する．各マルチプレクサの制御は，$n$ を用いて下位から $n-1$ ビット目以上は 1，$n-2$ ビット目以下は 0 となるような 31 ビットの信号を作る論理回路（**exp_s**）で行う．

### 5.2 モジュール combine の構成

モジュール **combine** は指数部と仮数部を結合し，丸めを行うモジュールである．ブロック図を図 6に示す．

結合は，まず，プライオリティエンコーダ **prienc_c** で **prienc_s** と同様に，31 ビット長の「指数値」の最上位ビットからの'0' または'1' の連続の個数を半分づつ数えて，エンコードしておく．この連続の個数を $m$ とすると，「指数部」のビット長 $l$ は $l = 63-2m$ である（表1）．次にこの「指数値」とケチ表現の 1 を取り除いた仮数値を繋げておき，シフタ **shifter_c** で $31-l = 31-(63-2m) = 2m-32$ ビット左シフトしたものの上位 31 ビットを切り出す．この構成は，**separate** と同様にして **prienc_c** で'0' または'1' の連続の個数をビット列を半分づつにして数えていき，**shifter_c** にバレルシフタを使用して，$2m-32$ ビットシフトする際に，先に決定される $m$ の上位ビットから使用することで遅延をオーバーラップさせることができる．それから指数値を指

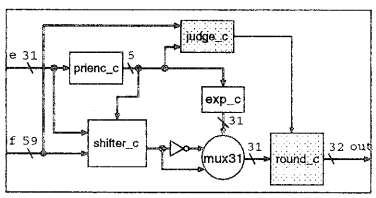


図 6: モジュール **combine** のブロック図

数部に変換する．これは，先の「切り出された 31 ビット」のビット列の上位から $n+1 = 33-m$ ビットを反転させればよい．構成は **separate** の場合と同様にして，「切り出された 31 ビット」の各ビットを反転させたものを作っておき，それを 31 個のマルチプレクサで選択する．各マルチプレクサの制御は上位から数えて $33-m$ ビット目以上が 1，$34-m$ ビット目以下が 0 となるような 31 ビットの出力をする論理回路（**exp_c**）で行なう．この場合，クリティカルパスを考えると，「31 ビットに切り出すためのシフト」と「指数値から指数部のビットパターンへの変換」が，順番に直列にくるため，**separate** と比べると，遅延時間が増える．しかし，乗算回路であることを考えると，**prienc_c**，**exp_c** の遅延は（乗算器）−（加算器）の遅延時間の部分にオーバーラップさせることができる．更に，「指数値から指数部への変換」は指数値だけでおこなえることから，「31 ビットに切り出す」前にこの処理を行えば，NOT ゲートとマルチプレクサ（**mux31**）の遅延もオーバーラップさせることができる．構成は，変換を行う **exp_c**，NOT ゲート，マルチプレクサをシフトの前に持って来て，**exp_c** を $m+1$ ビット目以上が 1，それ以下が 0 となるような 31 ビットの出力をする論理回路とすればよい．しかし，この構成では **prienc_c** と **shifter_c** とのオーバーラップができないため，指数と仮数の演算時間があまり違わない加算回路の場合では先の構成とする方がよい．以上のことから「結合」は「分離」とほぼ同じぐらいの遅延時間となる．

次に丸めについて述べる．丸めは図 6の網目のモジュールで行う．IEEE 規格の nearest モードで考える．丸めはまず，**judge_c** で，先のプライオリティエンコーダによりエンコードされた指数部のビット長の値 $l = 63-2m$ から，仮数値の下限を調べ，丸めの判定を行う．$31-l$ が仮数部のビット長であるから，仮数値の最上位ビットから $31-l$ ビット目が仮数部の下限である．この下限のビットから順に $p_0$，$r$ とし $33-l$ ビット目以下の論理和を $s$ として nearest モードの判定式（$(r \vee p_0) \wedge (r \vee s)$）で丸めの判定を行う．この判定は上の結合処理と並列に行え，また，遅延が **shifter_c** に比べて小さいため遅延時間の増加はない．次に上の判定に基づいて **round_c** で 1 を足す．1 を足す場合，IEEE 規格では仮数部の 23 ビットに 1 を足せばよいが，URR では再度分離するのは非効率的であり，また，分離しても 30 ビット長の仮数部に 1 を足すことになるため，32 ビット長のビット列全体に 1 を足した方が効率がよい．こうすれば，桁上がりが指数部に伝搬し，2 度目の正規化を必要としない．丸めを行わなければ **round_c** 分の遅延を減らすことができる．

## 6 「符号なし」版モジュールの性能

モジュール **urr_mul** の各モジュールの性能を表 2に示す．表 2は実装した回路と，それを最適化した回路の性能である．配線遅延は含まれていない．実装した回路の性能は，最適化したものに比べて，かなり冗長なものとなっている．最も回路規模の大きかったモジュール **mul** はア

表 2:「符号なし」版モジュールの性能

| モジュール | 実装した回路 | | 最適化した回路 | |
|---|---|---|---|---|
| | 面積 ($\mu m^2$) | 遅延 (ns) | 面積 ($\mu m^2$) | 遅延 (ns) |
| **SEPARATE** | 356,972 | 4.20 | 350,860 | 3.00 |
| **COMBINE** | 746,172 | 8.68 | 428,413 | 4.00 |
| **separate** | 297,199 | 1.99 | 251,317 | 2.00 |
| **combine** | 686,399 | 6.47 | 264,036 | 3.00 |
| **comp** | 41,992 | 1.99 | 51,755 | 1.00 |
| **normal** | 85,969 | 2.00 | 112,958 | 1.00 |
| **adder** | 102,877 | 2.89 | 96,685 | 2.00 |
| **mul** | 4,587,211 | 6.63 | – | – |
| **mux** | 17,781 | 0.22 | 17,781 | 0.22 |
| **nan** | 39,055 | 4.54 | – | – |
| **urr_mul** | 6,293,009 | 22.60 | – | – |

ルゴリズムに冗長 2 進加算木 [5] を用いた乗算器で面積は全体の約 73%を占めた．また，実装に用いた回路によるセルの使用頻度で重みをつけた，セル 1 個あたりの面積は約 $294\mu m^2$，遅延時間は約 0.13ns であった．

一方，最適化した回路では，最適化が合成ツールの性質に左右されることから，大きく分けて二通りの方針で最適化を行った．一つは，回路構成を詳細に検討して，その構造による記述を作成して合成する方法である．もう一つは，その動作をする回路の記述を，できる限りシンプルに動作記述したものを作成し，合成する方法である．例えば，**separate** の仮数部を取り出す部分は関数 **f** を用いて次のようにほぼ case 文一文で記述している．

```
casex(p_in[30:0])
31'b10?????????????????????????????:
  f={1'b1,in[28:0]};
31'b110????????????????????????????:
  f={1'b1,in[27:0],1'b0};
31'b1110???????????????????????????:
  f={1'b1,in[25:0],3'b0};
(以下略)
```

in[30:0] が入力で値，条件文の p_in[30:0] は 1 の連続のみを数えるようにしている．in が'0' の連続の場合にビット反転させる．指数値の取り出しは関数が返す値を拡張すればよい．この両者はどちらも動作記述レベルの記述であるため厳密には合成ツールの性能に依存しているが，前者の場合，合成ツールは記述された回路構成を基にして，その細部を最適化する傾向にあり，ある程度その構成が保持される面があるため，この回路構成を意識した記述で合成した回路の性能と比較することで，後者のシンプルな記述で合成した回路の性能をある程度見極めることができる．最も最適な回路を求めることは難しいが，以上のように比較検討することで，モジュールの性能の限界に近づくことができる．**separate, combine** の最適化はこのようにして行った．その結果，シンプルな記述により得られた回路の性能が僅かによかった．これは，合成ツールがライブラリの約 400 個ほどあるセルの性能を見ながら合成しているためゲートレベルでの最適化がなされやすいこと，使用した合成ツールのコンパイル能力が高いこと，モジュールが組み合わせ回路であるため，合成ツールによる圧縮が効率的に行われたことなどが考えられる．**comp, normal, adder** は基本的には加算回路である．加算は Verilog 演算子'+' で記述し合成ツールにより作成した桁上げ先見加算器であるが，4 ビット毎の桁上げ先見回路で構成したものより性能がよいことを確認している．

また，回路は遅延時間により面積が変化し，面積と遅延時間が反比例する傾向にある．モジュールを評価する際には，この曲線のどの点を取るかが問題となる．直観的には原点に最も近いものがよいと考えられるが，これではスケールの取り方に影響され一定でない．そこで,「面積と遅延時間の積」により性能を評価した．積が小さい程性能がよい．厳密ではないが，経験的には反比例より傾斜がきつい（$x^n y = const, (n \geq 1)$）ため，面積と遅延時間の割合が極端なものよりバランスがよいものの方が積が小さくなる傾向がある．このことから，直観的にも性能がよいと感じられるもの選ぶことができる．表 2はこのようにして幾つかの合成データから選択した数値である．

## 7 「符号あり」版 URR 乗算回路の構成

試作では「符号なし」版を用いたが，本来は分離/結合回路において 2 の補数をとる必要がない「符号あり」版で設計すべきである．以下,「符号なし」版との違いについて図 1の各構成要素毎に述べる．

分離回路では，仮数値の切り出しは先に検討した **separate** の構造とほとんど同じである．**separate** で切り出された仮数部のビット列は分離前に全体の 2 の補数を取らなければ 2 の補数であるから，演算を符号あり乗算で行えばそのまま扱える．仮数値の決定の速度がクリティカルパスであるから遅延時間は **separate** とほぼ同等である．一方，指数値の取り出しは，シフトして切り出された指数部から対応する指数値に変換する必要があり，また，表す値が負の場合の変換処理が追加される．しかし，負の場合は先の **separate** の構成における **exp_s** で，下位から数えて $n-1$ ビット目以下を 1 とすればよい（表 1）ため，**separate** よりも僅かに回路量が増加するのみである．

乗算器では，仮数部が 2 の補数で表現されているため，ケチ表現を付け加える際に，値が正であれば 01 を，負であれば 10 の 2 ビットを指数部の上位ビットに付け加える．このため，乗算器は符号あり 31 ビット乗算となる．

正規化では $f$ を「仮数値」とすると，URR における正規化の規則，$1 \leq f < 2, -2 \leq f < -1$ より，乗算結果は $-4 < f \times f < -1, 1 \leq f \times f \leq 4$ であるため，シフトする数の場合分けが「符号なし」版より増えることから回路量が少し増加する．

結合回路では，結合を行う部分は分離回路と同様，指数値から指数部への変換で値が負の場合を実装しなければならないが，負の場合は先の分離回路と同様にして **exp_c** を構成すればよいから **combine** に比べて回路量が僅かに増える程度である．丸めは IEEE 規格の nearest モードで考えると,「符号なし」版と同様である．IEEE 規格では判定の論理式 $((r \vee p_0) \wedge (r \vee s))$ が真となった場合に 1 を仮数に加算する.「符号なし」版では仮数値が正の場合しか

ないので，同様にして同じ論理式で判定できる．「符号あり」版では仮数値が2の補数表現されているが，2の補数に対するnearestモードの判定の論理式は先の正の場合と同じである．つまり，先の論理式を「符号あり」版の乗算結果に適用した場合の判定と，ビットを2の補数として解釈してnearestモードの意味（基本的には四捨五入で境界の場合には切り捨てると奇数となる場合だけ切り上げて偶数にする）を考えた場合の結果とが一致するため，「符号あり」版でも同じ論理式で判定できる．1の加算ではURRでビット列全体を解釈した場合の値の大小関係が，それを2の補数として見た場合と同じであるため「符号なし」版と同様ビット列全体に1を加算すればよい．

## 8 URRとIEEE規格の比較

IEEE規格との比較のため，URRとIEEE規格の各々で乗算回路を設計した．URRについては，「符号あり」版である．図1と対応させたIEEE規格の乗算回路の構成を図7に示す．IEEE規格では非正規化数の処理（漸進アンダーフロー）が必要であるが，非正規化数は頻繁に現れるものではないことから，一般的には例外処理としている[1]ためここでは考えない．丸めは両方ともIEEEのnearestモードとしている．

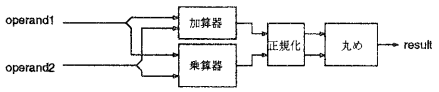


図7: IEEE規格の乗算回路の基本構成

表3に各構成要素の対応表と対応するモジュール名を示す．以下，各構成要素毎にそれぞれ比較検討する．分離回路はURRでは必要であるが，IEEE規格では対応するモジュールはない．クリティカルパスである仮数の乗算を行う乗算器では，IEEE規格の場合，仮数はゲタを加えた24ビットであるから，24ビット同士の乗算器で良いのに対し，URRでは，切り出した仮数値は31ビットであるから，31ビット同士の乗算器を必要とする．この乗算のビット長の違いは，URRがIEEE規格に比べて回路量が増える主な要因である．指数の加算を行う加算器では，IEEE規格の場合，指数は8ビットであるから，8ビット同士の加算器で良いのに対し，URRでは，切り出した指数値は

表3: 構成要素の比較

| 構成要素 | IEEE | URR |
|---|---|---|
| 分離回路 (Sep) | 必要なし (–) | 必要あり (**separate_urr**) |
| 乗算器 (Mul) | 24ビット乗算 (**mul24_ieee**) | 31ビット乗算 (**mul31_urr**) |
| 加算器 (Adder) | 8ビット加算 (**adder8_ieee**) | 31ビット加算 (**adder31_urr**) |
| 正規化 (Normal) | 8ビット加算 (**normalize_ieee**) | 31ビット加算 (**normalize_urr**) |
| 結合回路 (Comb) | 丸めに対応 (**rounding_ieee**) | 必要あり (**combine_urr**) |

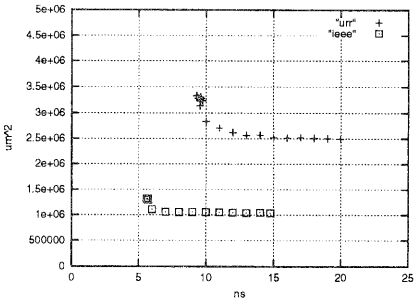


図8: URRとIEEEの性能比較結果（乗算）

表4: 遅延時間が最小なものによる比較（乗算）

| モジュール | 遅延 (ns) | 面積 ($\mu m^2$) | 遅延比 | 面積比 |
|---|---|---|---|---|
| **ieee** | 5.59 | 1,322,153 | 1.00 | 1.00 |
| **urr** | 9.30 | 3,332,134 | 1.66 | 2.52 |

31ビットであるから，31ビット同士の加算器を必要とする．よって，IEEE規格の方がURRに比べてビット数が小さいため遅延の小さい加算器になる．しかし，指数の加算はクリティカルパスではないため，URRで面積が少し増える程度である．正規化での処理は，仮数の左シフトと指数への加算であり，遅延が大きいのは指数への加算である．IEEE規格の場合，指数部が8ビットであるから，8ビットの加算で遅延時間が決まるのに対し，URRでは，指数部が31ビットであるため，31ビットの加算で遅延時間が決まるためIEEE規格に比べ遅延時間が大きくなる．結合回路は主に「結合」と「丸め」，それに伴う「2度目の正規化」を行っている部分である．IEEE規格でこれに対応する部分は，「丸め」と「2度目の正規化」である．URRで丸めを行わなければ，IEEEでの対応する回路はなく，URRでは約1nsの遅延時間短縮となる．

### 8.1 URRとIEEE規格の性能比較結果

図8，表4にURRとIEEE規格の性能比較結果を示す．図中，**urr**がURR乗算回路，**ieee**がIEEE規格の乗算回路である．乗算器はURRとIEEEではビット長が主な違いであるから比較の正確さのため，URR，IEEEで同一のアルゴリズムに固定している．乗算器は2次のbooth法とWallace-Treeを組合せたものを用いている．データは全てのモジュールの階層を壊して合成しているため，モジュール間の遅延オーバーラップは行なわれており，また冗長な論理等のロスは取り除かれている．表4は図8中遅延時間が最小なものによる比較である．面積に比べて遅延時間が重視される傾向にあることから，全体では遅延時間が最小なもので比較した．

### 8.2 モジュールの性能比較結果

乗算器を除いた各モジュールの性能比較を図9上に示す．図はできるかぎり最適化した回路である．図中**adder32**は32ビットの桁上げ先見加算器である．**adder32**を基準として比較すると，**separate_urr**では遅延時間はあま

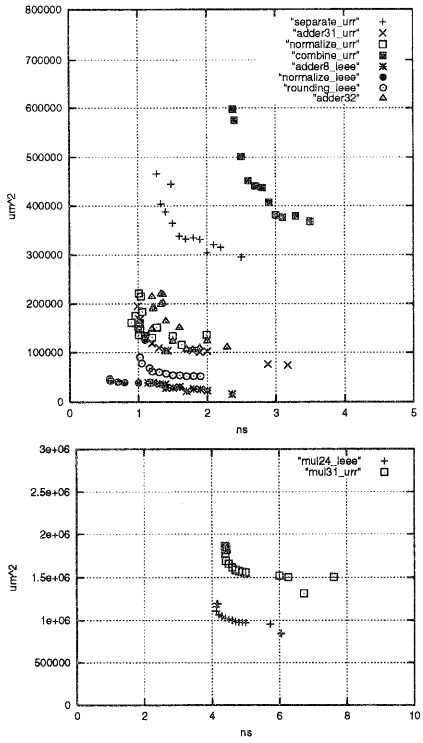


図 9: 各モジュールの性能：乗算器を除く (上) 乗算器 (下)

り違わないが，面積が倍以上になっている．これは，分離の処理でバレルシフタに加えてプライオリティエンコーダが必要であるためである．**combine_urr** では丸めも行っているため，更に 1 の加算回路だけ回路量が増えている．**normalize_urr** は 31 ビットに 1 又は 2 を加算する回路と，判定の回路であり，**adder32** に比べて遅延時間で少し小さい．**adder31_urr** は 31 ビット加算器でほぼ同じである．**rounding_ieee** は丸め判定，丸め（23 ビットの 1 加算），2 度目の正規化（8 ビットの 1 加算）が順に行われるもので，**adder32** に比べて面積が半分程度である．**normalize_ieee** は 8 ビットの 1 加算であるためかなり小さい．**adder8_ieee** は 8 ビットの加算器であり面積はかなり小さいが，ゲタを引く分だけ遅延は僅かに増えている．図 9 下に乗算器の性能比較を示す．乗算器はどちらも 2 次の booth 法と Wallace-Tree を組合せたものである．2 次の booth 法は部分積を通常に比べ半分に減らす方法である．この組合せは高速乗算器のアルゴリズムの中では比較的性能が良い．図中，**mul24_ieee** が IEEE の 24 ビット長乗算器，**mul31_urr** が URR の 31 ビット長乗算器である．

### 8.3 構成要素毎の比較結果

図 10に各構成要素毎の比較結果を示す．図 9上，図 9 下の各モジュールのデータの中から積が最小なもの選択して，そのモジュールを対応する構成要素としている．Adder

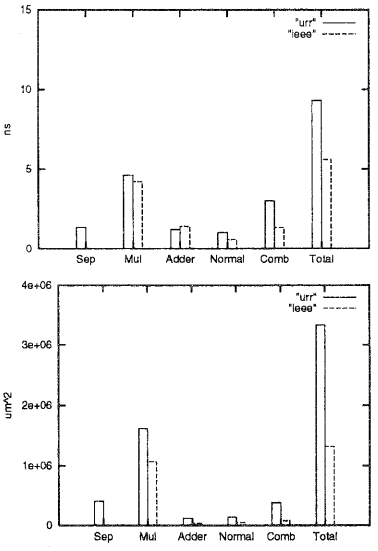


図 10: 構成要素毎の比較結果：遅延時間 (上) 面積 (下)

(**adder8_ieee**, **adder31_urr**) は配列型乗算器に比べて遅延が小さくなる程度に面積を優先させた桁上げ先見加算器とした．Total は合計である．遅延は Total = Sep + Mul + Normal + Comb，面積は Total = (Sep×2)+ Mul + Adder + Normal + Comb である．図ではモジュール間の冗長成分が取り除かれているためそれよりも小さくなっている．図 10上より遅延時間では Mul が最も大きい．比較では Sep，Mul，Comb の差が大きいことが分かる．Adder は大きいように見えるがクリティカルパスでないため Total に加算されず問題ない．Total での差は 1.66 倍であった．図 10下より面積でも Mul がほとんどを占めている．IEEE 規格では Total の面積が Mul とほぼ等しい．比較では Sep，Mul，Comb で差が大きく，その他はあまり差がない．Total での差は 2.52 倍であった．分離回路が 2 つ必要であることや，乗算器のビット長の違いが面積では差が開いたこと，Comb の差が面積では大きいことなどで遅延時間の Total に比べて差が大きくなっている．

## 9 加算回路の比較

加算回路の性能比較のため，乗算回路の場合と同様にして，IEEE 規格と URR とでそれぞれ浮動小数点加算回路を Verilog-HDL を用いて設計し，DesignCompiler を用いて合成した．合成結果を図 11，表 5に示す．表 5は図 11 中遅延時間が最小なものである．乗算回路と比べると遅延時間が少し大きくなっている．IEEE の場合を考えると，加算回路での桁合わせの仮数のシフト，仮数の加算，加算結果が負の場合の 2 の補数をとる回路，正規化での仮数のシフトが，乗算回路の乗算器に対応している．乗算では乗算器に比較的高速なアルゴリズムを採用していることもあり，

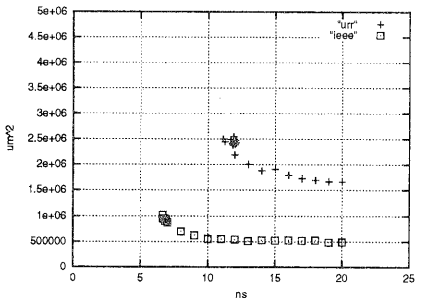


図 11: URR と IEEE の性能比較結果（加算）

表 5: 遅延時間が最小なものによる比較（加算）

| モジュール | 遅延 (ns) | 面積 ($\mu m^2$) | 遅延比 | 面積比 |
|---|---|---|---|---|
| **ieee** | 6.63 | 1,020,747 | 1.00 | 1.00 |
| **urr** | 11.13 | 2,495,548 | 1.68 | 2.44 |

僅かに遅延時間が大きくなった．URR 加算回路では上の 4 つの操作の中の「加算結果が負の場合の 2 の補数をとる回路」が必要ないが，各々の操作でビット長が長いために乗算回路の場合と同程度の差が出ている．

## 10 試作チップ

試作チップ[1]の主な製造条件は，CMOS 0.6$\mu$m，PolySi2 層，メタル配線 3 層，チップサイズ 4.5mm 角，信号ピン数 87，である．設計した URR 乗算回路は 32 ビットであるため，$32 \times 3 = 96$ の IO ピンが必要であるが，試作チップの IO ピン数（87）の制約により，IO ピンを時分割で用いている．合成結果は，21,446 セル使用し，インバータ換算で 39,880 であった．配置配線で作成したレイアウトを図 12に示す．単一階層でレイアウトを作成している．結果は，Core 部が配置可能な領域の約 68%を占めた．図 12より，設計がチップサイズに対し，比較的大きなものであることが分かる．試作チップは現在測定中であるが，幾つかのデータを与えた結果，動作することを確認している．

## 11 おわりに

本論文では，主に回路量の視点から乗算回路の場合における URR と IEEE 規格との比較について述べた．その結果，遅延時間で 1.66 倍，面積で 2.52 倍となった．乗算回路であることから，乗算器が大きいこともあり遅延時間ではあまり差はでなかった．面積では差が 2 倍以上と大きくなった．更に加算回路の場合の比較についても示し，遅延で 1.68 倍，面積で 2.44 倍となった．

IEEE では仮数が絶対値，URR では 2 の補数であるが，これによる実装上の得失は，加算回路の場合に URR で仮数の加算結果が負となるときに 2 の補数を取る必要がない分だけ若干効率が良いが違いはほとんどなかった．また，

[1] 本チップ試作は東京大学大規模集積システム設計教育研究センターを通しローム（株）および凸版印刷（株）の協力で行われたものである．

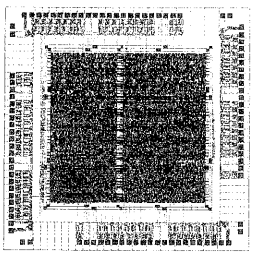

図 12: 作成したレイアウト

URR の大きな特徴は，指数部と仮数部が可変長であるために絶対値の大きな値や小さな値を表すことができる点であるが，その境界情報を 0 または 1 の連続の数が持っていることから，分離/結合でプライオリティエンコーダとシフタという操作を必要とする．境界情報を 0 または 1 の連続の数に持たせるのではなく始めから境界位置情報のフィールドを作っておきそこに数値として置いておけば [6]，プライオリティエンコーダを必要とせずシフトのみで良いため，広い範囲の値を表現できるという性質を持ったまま分離/結合回路を更に高速にすることができる．しかしこの場合は，URR の性質である，長さの異なるデータ間の相互変換が容易であるという長所 [2][3] を利用できなくなる．

URR は表せる値の範囲が大きいことから，計算過程で数値が大きく変化するような数値計算などには有効である．本論文では，URR の評価のための一つの材料として乗算回路/加算回路の性能評価を示した．

**謝辞** 本論文をまとめるにあたり，各種 CAD ツール，セルライブラリを提供して頂いた VDEC 及び関係者の方々に深く感謝します．

回路構成について多くの貴重な助言を頂いた，電気通信大学中川圭介教授，鶴田三敏氏（現（株）LSI システムズ）に深く感謝します．

## 参考文献

[1] J.L.Hennessy, D.A.Patterson: "Computer Architecture:A Quantitiative Approach, 2nd ed.", Morgan Kaufmann Pub. Inc., 1996.

[2] Hamada.H: "A New Real Number Representation and Its Operation", Proc. 8th Symposium on Computer Arithmetic, pp.153–157, 1987.

[3] 浜田穂積: "二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.

[4] D.E.Thomas, P.R.Moorby: " The Verilog Hardware Description Language, 2nd ed.", Kluwer Academic Pub., 1995.

[5] 高木直史, 安浦寛人, 矢島修三: "冗長 2 進加算木を用いた VLSI 向き高速乗算器", 電子通信学会論文誌, Vol.J66-D, No.6, pp.683–690, 1983.

[6] 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.