Memoirs of the Faculty of Science
Kochi University (Information Science)
Vol. 26 (2005), No. 1

# 32 ビット浮動小数点プロセッサの機能検証と改善
# （加減算器・メモリ回路）

中薗佳寿　増田智一　國信茂郎

高知大学理学部数理情報科学科

Abstract

近年の集積回路の高集積度による VLSI（大規模集積回路）の出現により、HDL（ハードウェア記述言語）を用いた機能設計からトップダウン設計が重要になってきている。

本研究では、浮動小数点プロセッサ(Floating Point Unit)の加減算器の機能検証を行い、加減算器が完全でないところを修正し、新しい加減算器を設計しなおした。また、メモリ回路の基本動作について記述した。

# 序論

我々の生活環境にある電気・電子機器にはLSIが多数搭載されている。例えばパソコンやゲーム機にはCPUやメモリ、さらには周辺機器制御用LSIなどの様々なLSIが搭載されている。このような状況が出現したのは、半導体のプロセス技術の進展により LSI に搭載できる回路の規模が増大し、高度で多様な機能を実現できるようになったからである。

その背景になるコンピュータの中核テクノロジの世代区分を説明すると、第１世代には真空管が使われ、第２世代に入るとトランジスタが使われた。トランジスタとは電気的な ON/OFF 動作をするスイッチである。第３世代に移ると集積回路すなわちチップが登場する。集積回路は数十から数百のトランジスタを１チップにまとめて接続したものである。その後のトランジスタ数は、数百から数百万へと驚異的に増大した。このような集積回路のことを超大規模集積回路と呼ぶ。この超大規模集積回路が現在の主役であり、略して VLSI と呼ぶ。

ここで世代ごとに単位コスト当たりの相対性能比を比べてみる。第１世代の値を１とすると、第２世代は３５、第３世代は９００、第４世代は２,４００,０００、と驚異的に高性能化されている。最も回路が大規模で複雑なものになれば回路設計の困難は増大する。そこで LSI の大規模化・複雑化に対応する設計技術にも革新が起こった。すなわち、RTL 回路を HDL と言うハードウェアを設計するための言語で記述し、LSI をトップダウン的に設計する手法が開発され、広く用いられるようになった。

ここで設計技術の進展を見てみると LSI 設計のシミュレーション技術は、1980 年代に入りゲートレベルシミュレーションが自動化され、ボード上でのシミュレーションからコンピュータ上でのシミュレーションが可能になった。しかし、近年の設計規模の拡大により従来の設計方法である回路図入力ではゲートレベルシミュレーションを用いても大規模回路を要求しよう通りに短期間で開発することが困難になってきたので設計時間の短縮とコスト削減の要求性、また優れた論理合成ツールの出現によりゲートレベル設計から HDL 記述設計へ変異している。

HDL とは回路の動作や構造を記述するための言語で、回路設計では従来のように回路図入力ではなく HDL によって回路動作を記述する。HDL は主に C 言語のような記述性をもつ Verilog-HDL と IEEE Std-1076 規格に基づく VHDL がある。HDL の出現により、以前までのセル部から設計を始め、最終的にターゲットの製品を作成する方法（ボトムアップ設計）から、機能の検証からアプローチする方法（トップダウン設計）に移行している。トップダウン設計では設計工程を機能検証工程とゲートへの具体化作業工程の二つに分けることにより、設計者ははじめからタイミングなどの考慮をする必要がなく、機能設計に関するバグを早い段階で見つけることができ、開発期間を短縮している。

本研究室では、昨年 HDL（ハードウェア記述言語）を用いた浮動小数点プロセッサの設計を行った。私たちの研究はこれを継続したものである。

具体的には、昨年の先輩方が HDL（ハードウェア記述言語）を用いて設計された３２ビット

浮動小数点プロセッサ（FPU）の加減算器（ALU）の誤り箇所を修正し、正しい加減算器の設計することと、３２ビット浮動小数点プロセッサ（FPU）のマルチポートレジスタ（MPR）で使用されるメモリ LSI の回路構造をより理解することにとした。

本論文では、１章で、昨年の論文の修正を行い、２章でメモリ LSI の回路設計を行うこととする。

# １　加減算器（FAU）の機能検証と改善

この章では、昨年の先輩方が設計されたFPUの加減算器の修正を行う。次から、“浮動小数点加減算器の概要”、“加減算器のシミュレーションによるバグ出し”の後、“加減算器の修正”を行っていく。

## １．１　浮動小数点加減算器の概要

ここでは、まず“浮動小数点とは”、“32ビット浮動小数点数の表現形式”、“浮動小数点加算の概略”で、浮動小数点と浮動小数点加算の説明をします。次に、“加減算器の設計”、“加減算器のVerilogHDLによる記述”で、浮動小数点加減算器の説明をしていく。

### １．１．１　浮動小数点とは

符号なし整数および符号付き整数の他に、プログラミング言語では小数を含む数値を取り扱う。そのような数値を実数（real）形式という。例えば以下のようなものである。

３．１４１９５２６５...

１．０×１０$^{-9}$

特に２つ目の例に使われている表記法は科学表記法と呼ばれ、コンピュータで扱うときには、浮動小数点形式と呼ばれる。通常、浮動小数点形式の数を書き表す場合、小数点の左側には数字を１つしか書かない。一般形は以下のようになる。

１．ｘｘｘｘｘｘｘｘｘ$_{(2)}$ ×２$^{yyy}$

実数を標準的な科学記数法の正規形で書き直す利点は3つある。１つ目には、浮動小数点数を含むデータの交換が単純化される。２つ目には、浮動小数点演算アルゴリズムが単純化される。3つ目は、１語に収納できる数値の精度を上げることができる。それは、不必要な先行する０の代わりに、小数点の右側に実数を記すことができるからである。

## １．１．２　３２ビット浮動小数点数の表現形式

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | …… | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| s | 指数 | | | | | | | | 仮数 | | | | | | | | | |

１ビット　　８ビット　　２３ビット

表１．１　IEEE754 での浮動小数点数の表現形式（単精度）

- S は浮動小数点数の符号を表す（１は負を意味する）。
- 指数は８ビットの指数フィールドを表す（指数の符号を含む）。・・・(*1)
- 仮数は 23 ビットの小数を表す。

(*1) 最も小さな負の指数を 00000000 と表し，最も大きい正の指数を 1111111 と表すゲタばき表現をとる。（IEEE754 では単精度用のゲタに 127 を使用。）

－１ ⇒ －１＋１２７(10)＝１２６(10) ⇒ ０１１１１１１０(2)

＋１ ⇒ ＋１＋１２７(10)＝１２８(10) ⇒ １０００００００(2)

以上のように指数にゲタをはかせた場合、浮動小数点数の実際の値は下記の式によって表されることになる。

$(-1)^{s} \times (1 + \text{仮数}) \times 2^{(\text{指数}-\text{ゲタ})}$

（＊）いくつかの 32 ビット浮動小数点数を１０進数に直してみた。

1 10000001 11000000000000000000000＝ -7(10)
1 10000001 10000000000000000000000＝ -6(10)
1 10000001 01000000000000000000000＝ -5(10)
1 10000001 00000000000000000000000＝ -4(10)
1 10000000 10000000000000000000000＝ -3(10)
1 10000000 00000000000000000000000＝ -2(10)
1 01111111 00000000000000000000000＝ -1(10)
0 00000000 00000000000000000000000＝ 0(10)
0 01111111 00000000000000000000000＝ 1(10)
0 10000000 00000000000000000000000＝ 2(10)
0 10000000 10000000000000000000000＝ 3(10)
0 10000001 00000000000000000000000＝ 4(10)
0 10000001 01000000000000000000000＝ 5(10)
0 10000001 10000000000000000000000＝ 6(10)
0 10000001 11000000000000000000000＝ 7(10)
0 01111101 00000000000000000000000＝0.25(10)
0 01111110 00000000000000000000000＝0.5(10)
0 01111110 10000000000000000000000＝0.75(10)

### １．１．３　浮動小数点加算の概略

２進の浮動小数点数を加算するアルゴリズムを図１．１に示す。ステップ１と２では指数の小さい方の数値の仮数を調整し、それから２つの仮数を加える。その結果をステップ３で正規化する。単純化を図るために、ステップ４では丸めを行う。丸めの方法としては(*1)、IEEE754 規格の４つのオプションのうち１つを使用する。浮動小数点演算の正確性は丸めの正確性に大きく影響される。丸めることは容易であるが、正確性が損なわれる原因となる。

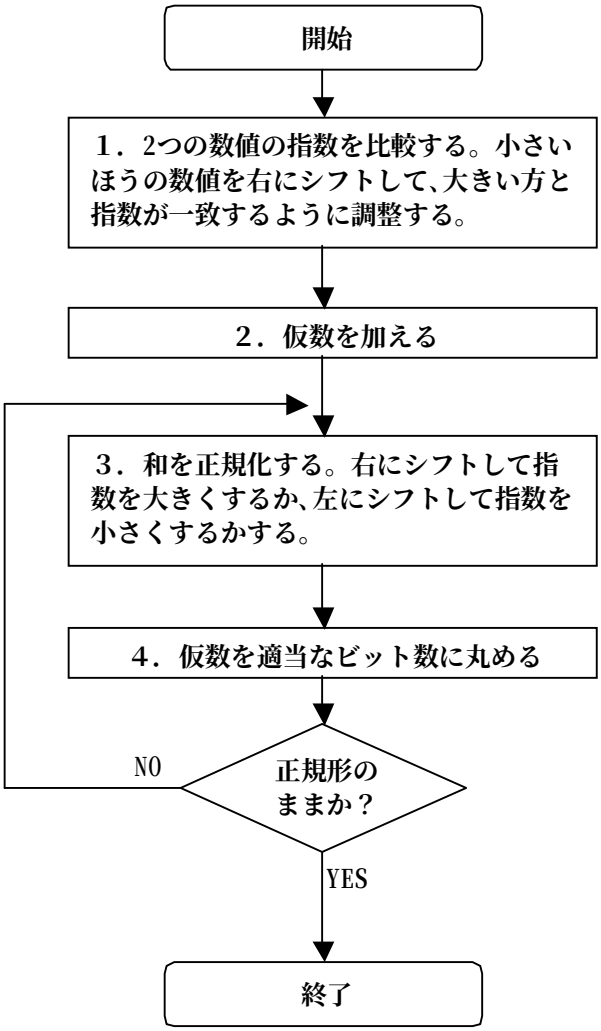


図１．１　浮動小数点加算のアルゴリズム

（*1）IEEE754 規格丸めモード

| | |
|---|---|
| RN(Round to Nearst) | 最も近い値に丸める |
| RP(Round to Plus) | ＋∞に向かって丸める |
| RM(Round to Minus) | －∞に向かって丸める |
| RZ(Round to Zero) | ０に向かって丸める |

＊本プロセッサでは RN（最も近い値に丸める）を使用

それでは簡単な科学表記法で表現された数値を手計算で加算して、浮動小数点加算における問題を明らかにする。では、$1.000_{(2)} \times 2^{-1}$と$-1.110_{(2)} \times 2^{-2}$を計算しよう。ここで、仮数には４桁、指数には２桁しか格納できないものと仮定する。

では、アルゴリズムの順を追って、計算を進めよう。

ステップ１：指数が小さい方の数値（$-1.110_{(2)} \times 2^{-2}$）の仮数を右にシフトして、大きい方の数値と指数が一致するようにする。

$$-1.110_{(2)} \times 2^{-2} = -0.111_{(2)} \times 2^{-1}$$

ステップ２：２つの仮数を加算する。

$$1.0_{(2)} \times 2^{-1} + (-0.111_{(2)} \times 2^{-1}) = 0.001 \times 2^{-1}$$

ステップ３：和を正規化する。

$$0.001_{(2)} \times 2^{-1} = 1.000_{(2)} \times 2^{-4}$$

ステップ４：和を丸める。

$$1.000_{(2)} \times 2^{-4}$$

この和は４ビットに収まっている。従って、丸める必要はない。

### １．１．４　加減算器の設計

以上のような手順で浮動小数点加算は行われる。また、浮動小数点減算についても同様の手順で行うことが出来る。以下では加減算器の設計について述べる。単精度浮動小数演算，さらに倍精度浮動小数演算へと演算のビット長が増加してくると、加算の桁上げ伝播による遅延が高速化を妨げるようになってくる。従来、桁上げ伝播の遅延を減らしたり、桁上げ伝播の遅延を無くす方法がいくつか提案されているようだ。遅延を減らす方法として桁上げ先見加算器があり，加算時間は高速であるが加算器を構成するのに必要な素子数（トランジスタ数）が多く，また、LSI として実現するためのレイアウトの規則性（容易性）の点からも課題が多い。従って、実際には桁上げ先見回路と順次桁上げ加算器の組み合わせによる加算器、あるいは桁上げ先見回路と桁上げ先見加算木の組み合わせによる加算器をはじめとする桁上げ先見回路を基本とする加算器がよく用いられる。また、一般に遅延を減らす方法として桁上げ保存法は２数の加算に適用すると１サイクル毎の逐次演算となり高速演算は実現できない。そこで今回、本加算器では，ブロック桁上げ先見回路と桁上げ先見加算木によるものを基本としている。

**（１）シフト量演算用の絶対値出力回路**

**図１．１に示した浮動小数点加算実行の第一ステップ、すなわち，２数の指数を一致させるための仮数の桁合わせの処理がある。通常、2 数の大小を比較し、多きい数から小さい数を減算するという作業で指数の差の絶対値を逐次的に求めることが出来る。その処理を図１．２に示した回路を用いて、２数の指数の差（絶対値）、および大きい方の指数を求めることが出来る。本回路では、2 数の指数部をｘ，ｙとして桁上げ入力を** c_1 **としている。**

**単位回路** A

s[i]=x[i]^~y[i]^ci-1

gi=x[i]&y[i]

pi=x[i]|y[i]

**単位回路** B

pi,i+3=p[i]&p[i+1]&p[i+2]&p[i+3]

gi,i+3=g[i+3]|(p[i+3]&g[i+2])|(p[i+2]&p[i+1]&g[i+1])

|(p[i+3]&p[i+2]&p[i+1]&g[i])

**単位回路** C

pi,k=pi,j&pj+1,k

gi,k=gi,j|(pj+1&gi,j)

cj=gi,j|(ci_1&pi,j)

**この回路は、２つの指数の差（**x-y**）を** c_1=1 **を用いることにより、その２の補数値** ~(x-(y+1))(=-(x-y))**を** c_1=0 **を用いることにより求めることが出来る。桁上げ入力** c_1 **の値は、**

c_1=g7|(c6&p7)・・・ci=gi|(ci_1&pi)**より**

**により求まる。また，この値は桁あふれ信号にも対応している。では実際に** x=10000000,y=01111100 **の時の絶対値を求めるとする。**

y1=10000011

p0=1,p1=1,p2=0,…..,p6=0,p7=1

g0=0,g1=0,…….,g6=0,g7=1

p0,3=0, g0,3=0

p4,7=0, g4,7=1

c_1=1,c0=1,c1=1,c2=0,c3=0,c4=0,c5=0,c6=0

s=00000100

**このとき、**c_1=1 **より**(x-y)=00000100 **が求まった。**

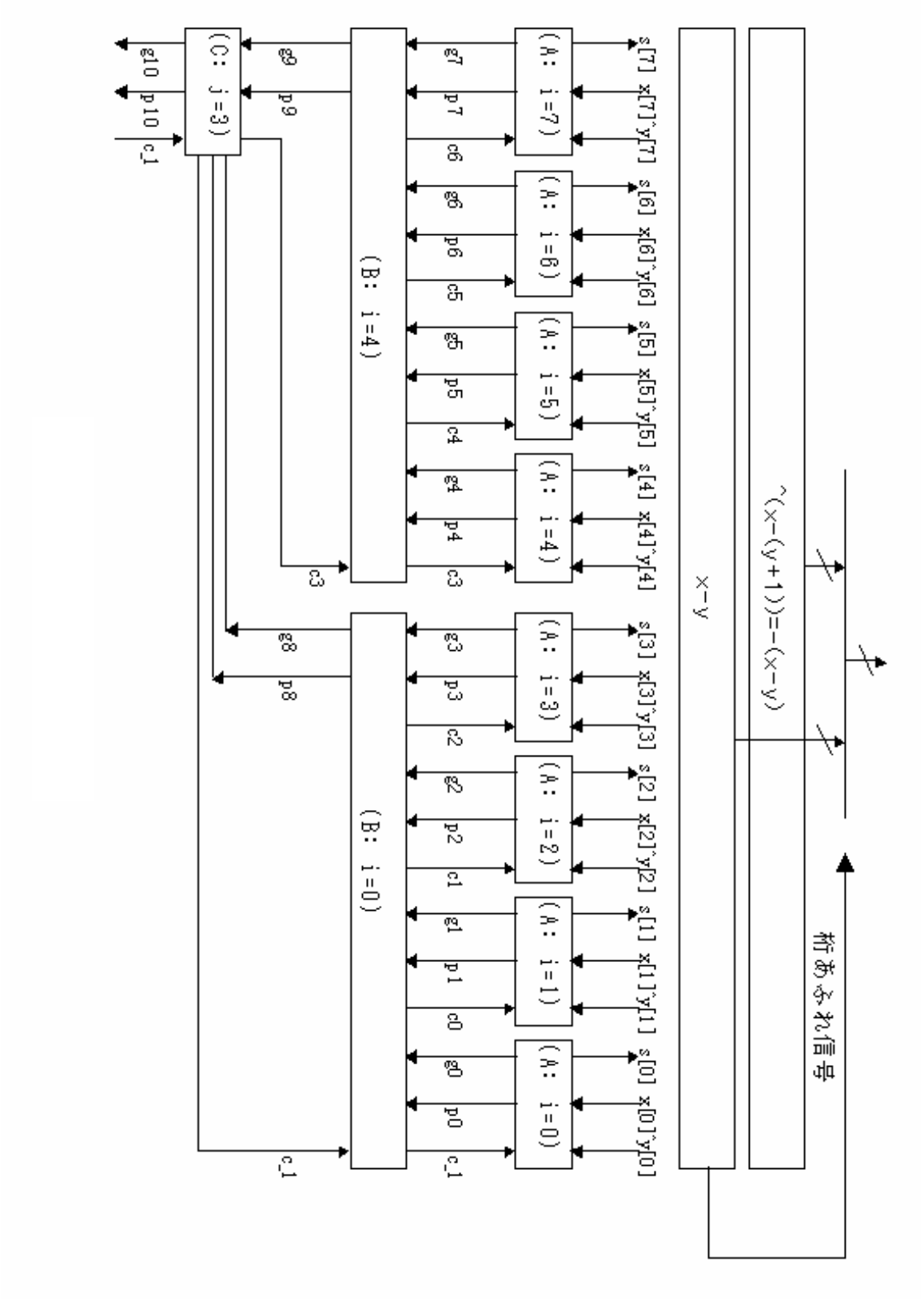


**図１．２　8 ビット絶対値出力回路**

（２）仮数部加減算回路

図１．１に示した加算の第二ステップ、即ち仮数の桁あわせの後の仮数の加算の処理がある。本浮動小数点プロセッサでは、シフト量演算用の減算絶対値出力回路と同様、ブロック桁上げ先見回路と桁上げ先見加算木による加算器を用いている。回路構成を図１．３に示す。図１．３は図１．２の絶対値出力回路の最下位ビットに２ビットの順次桁上げ加算器を付加した構成になっている。今仮に、単位回路Ａ（i=0）を仮数部の最下位ビットとし、丸めによる桁上げ（s[-1]と s[-2]）により答を得るものである。本浮動小数点プロセッサでは丸めモードに IEEE754 のＲＮ（最も近い値を求める）を使用しているので、{s[-1],s[-2]} の値が１０以上のときに桁上げを行うものとする。

なお、加算または減算処理によって桁落ちが生じる場合は、結果の上位ビットの０の数に応じてビットの左シフトを行う。このとき、最下位ビット側は０を入力するので演算精度は悪くなる。

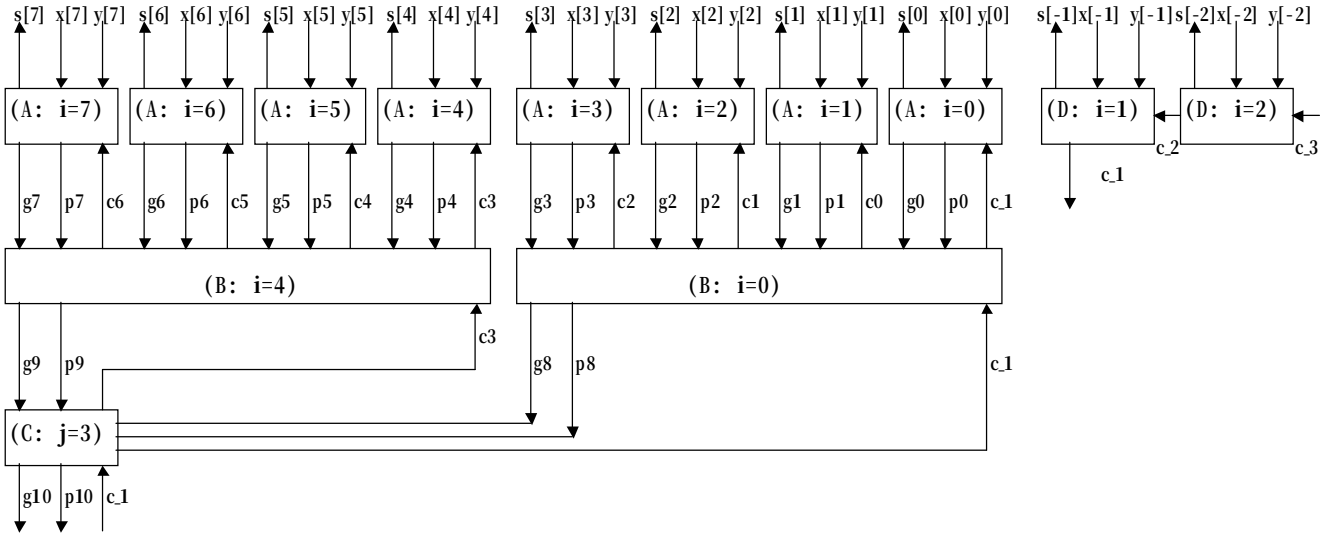


図１．３　４ビット長ブロック桁上げ先見回路と桁上げ先見加算木による加算器と丸め処理

## １．１．５　加減算器のVerilogHDLによる記述

### （1）機能

・FAUsel 制御信号に従って２つの入力 a,b を演算（加算または減算）し、out に出力する。
・FAUsel 制御信号による各種演算の選択は表１．２を参照。

### （2）シンボル図と各信号の機能

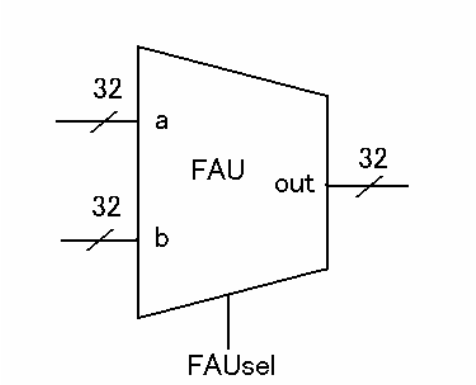


図１．４　加減算器（FAU）のシンボル図

| 信号 | 機能 | 補足 |
|---|---|---|
| a,b | ３２ビットの入力 | |
| out | ３２ビットの出力 | 演算結果の出力 |
| FAUsel | 制御信号 | 0 のとき加算、1 のとき減算 |

表１．２　FAU で使用する信号の機能

### （3）HDL 記述

```
/*加減算器の HDL 記述（上位モジュール）*/
module fau(a,b,FAUsel,out);
    input  [31:0]  a,b;
    input          FAUsel;
    outputt [31:0]  out;
    reg    [31:0]  out;
    wire   [4:0]   u;
```

```
 wire    [25:0]  k,l,o;
 wire    [7:0]   q;
 wire    [7:0]   p;
 reg     [22:0]  r,s;
 wire    [24:0]  v0;
 wire    [23:0]  v1;

always @(a or b)begin
   if(a[30:23]>b[30:23])begin
       s <= b[22:0];                          //小さな数値の仮数部（s）
       r <= a[22:0];                          //大きな数値の仮数部（r）
       out[31] <= a[31];                          //符号部（f）
   end else if(a[30:23]<b[30:23])begin
       s <= a[22:0];
       r <= b[22:0];
       out[31] <= b[31];
   end else if(a[30:23]==b[30:23])begin
       if(a[22:0]>b[22:0])begin
                s <= b[22:0];
                r <= a[22:0];
                out[31] <= a[31];
       end else if(a[22:0]<b[22:0])begin
                s <= a[22:0];
                r <= b[22:0];
                out[31] <= b[31];
       end
   end
end

zetaiti i0(.e1(a[30:23]),.e2(b[30:23]),.s(q),.p(p));  //指数の差(q) 大きな数値の指数部(p)

assign k = {1'b1,s,2'b00};          //丸め用に仮数部の下位2ビットを増やす
assign l = {1'b1,r,2'b00};          //先頭に1.・・・の1を付ける

assign o = k >> q;                  //小さな数値を指数の差だけ右にシフト
add i1(.x(l),.y(o),.s(v0));         //加算
sub i2(.x(l),.y(o),.s(v1));         //減算
```

```
function [4:0] discover_first_1;              //先頭から何桁目に1が来るかを判定
        input  [23:0]  t;
        if(t[23]==1'b1)  discover_first_1 = 5'b00000;
        else if(t[22]==1'b1)  discover_first_1 = 5'b00001;
        else if(t[21]==1'b1)  discover_first_1 = 5'b00010;
        else if(t[20]==1'b1)  discover_first_1 = 5'b00011;
        else if(t[19]==1'b1)  discover_first_1 = 5'b00100;
        else if(t[18]==1'b1)  discover_first_1 = 5'b00101;
        else if(t[17]==1'b1)  discover_first_1 = 5'b00110;
        else if(t[16]==1'b1)  discover_first_1 = 5'b00111;
        else if(t[15]==1'b1)  discover_first_1 = 5'b01000;
        else if(t[14]==1'b1)  discover_first_1 = 5'b01001;
        else if(t[13]==1'b1)  discover_first_1 = 5'b01010;
        else if(t[12]==1'b1)  discover_first_1 = 5'b01011;
        else if(t[11]==1'b1)  discover_first_1 = 5'b01100;
        else if(t[10]==1'b1)  discover_first_1 = 5'b01101;
        else if(t[9]==1'b1)  discover_first_1 = 5'b01110;
        else if(t[8]==1'b1)  discover_first_1 = 5'b01111;
        else if(t[7]==1'b1)  discover_first_1 = 5'b10000;
        else if(t[6]==1'b1)  discover_first_1 = 5'b10001;
        else if(t[5]==1'b1)  discover_first_1 = 5'b10010;
        else if(t[4]==1'b1)  discover_first_1 = 5'b10011;
        else if(t[3]==1'b1)  discover_first_1 = 5'b10100;
        else if(t[2]==1'b1)  discover_first_1 = 5'b10101;
        else if(t[1]==1'b1)  discover_first_1 = 5'b10110;
        else if(t[0]==1'b1)  discover_first_1 = 5'b10111;
endfunction

assign u = discover_first_1(v1[23:0]);

always @(v0 or v1 or p or u or FAUsel) begin
       if(FAUsel==1'b0)begin
            if(v0[24]==1'b1)begin                    //正規化(右シフト)
               out[22:0] <= v0 >> 1'b1;
               out[30:23] <= p + 1'b1;
            end else if (v0[24]==1'b0) begin
               out[22:0] <= v0;
```

```
                out[30:23] <= p;
            end
        end else if(FAUsel==1'b1)begin
            out[22:0] <= v1 << u;              //正規化（左シフト）
            out[30:23] <= p  - u;
        end
    end
endmodule


/*シフト量算出用絶対値出力回路（下位モジュール）*/
module zetaiti(e1,e2,s,p);
    input  [7:0]  e1,e2;
    output [7:0]  s,p;
    reg    [7:0]  s,p;
    wire   [7:0]  ee,ss;
    wire          c,c_1,c0,c1,c2,c3,c4,c5,c6,c7,c8,p0,p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,
                  g0,g1,g2,g3,g4,g5,g6,g7,g8,g9,g10;

    assign ee = ~e2;

    kairo_a i0(.e1(e1[0]),.e2(ee[0]),.c(c_1),.s(ss[0]),.p(p0),.g(g0));
    kairo_a i1(.e1(e1[1]),.e2(ee[1]),.c(c0),.s(ss[1]),.p(p1),.g(g1));
    kairo_a i2(.e1(e1[2]),.e2(ee[2]),.c(c1),.s(ss[2]),.p(p2),.g(g2));
    kairo_a i3(.e1(e1[3]),.e2(ee[3]),.c(c2),.s(ss[3]),.p(p3),.g(g3));
    kairo_a i4(.e1(e1[4]),.e2(ee[4]),.c(c3),.s(ss[4]),.p(p4),.g(g4));
    kairo_a i5(.e1(e1[5]),.e2(ee[5]),.c(c4),.s(ss[5]),.p(p5),.g(g5));
    kairo_a i6(.e1(e1[6]),.e2(ee[6]),.c(c5),.s(ss[6]),.p(p6),.g(g6));
    kairo_a i7(.e1(e1[7]),.e2(ee[7]),.c(c6),.s(ss[7]),.p(p7),.g(g7));

        kairo_b i8(.p({p3,p2,p1,p0}),.g({g3,g2,g1,g0}),.c(c7),.pp(p8),.gg(g8),.cc({c2,c1,c0,c_1}));
        kairo_b i9(.p({p7,p6,p5,p4}),.g({g7,g6,g5,g4}),.c(c8),.pp(p9),.gg(g9),.cc({c6,c5,c4,c3}));

    kairo_c i10(.c(c),.p({p9,p8}),.g({g9,g8}),.cc({c8,c7}),.pp(p10),.gg(g10));

    assign c = g10;             //assign c = g7  | (c6&p7);
```

```
    always @(ss)begin
        if(c==1'b1)begin
            s<=ss;
            p<=e1;
        end else begin
            s<= -ss;
            p<=e2;
        end
    end
endmodule


/*桁上げ先見回路による add 回路（下位モジュール）*/
module add(x,y,s);
    input  [25:0]  x,y;
    output [24:0]  s;
    reg    [24:0]  s;
    wire   [26:0]  ss;
    wire           c_3,c_2,c_1,c0,c1,c2,c3,c4,c5,c6,c7,c8,c9,c10,c11,c12,c13,c14,c15,c16,c17,
                   c18,c19,c20,c21,c22,c23,c24,c25,c26,c27,c28,c29,c30,c31,c32,p0,p1,p2,p3,p4,
                   p5,p6,p7,p8,p9,p10,p11,p12,p13,p14,p15,p16,p17,p18,p19,p20,
                   p21,p22,p23,p24,p25,p26,p27,p28,p29,p30,p31,p32,p33,p34,
                   g0,g1,g2,g3,g4,g5,g6,g7,g8,g9,g10,g11,g12,g13,g14,g15,g16,g17,g18,
                   g19,g20,g21,g22,g23,g24,g25,g26,g27,g28,g29,g30,g31,g32,g33,g34;
     reg            c;

    kairo_d i35(.x(x[0]),.y(y[0]),.c(1'b0),.cc(c_3),.s(ss[0]));
    kairo_d i36(.x(x[1]),.y(y[1]),.c(c_3),.cc(c_2),.s(ss[1]));

    kairo_a i0(.e1(x[2]),.e2(y[2]),.c(c_1),.s(ss[2]),.p(p0),.g(g0));
    kairo_a i1(.e1(x[3]),.e2(y[3]),.c(c0),.s(ss[3]),.p(p1),.g(g1));
    kairo_a i2(.e1(x[4]),.e2(y[4]),.c(c1),.s(ss[4]),.p(p2),.g(g2));
    kairo_a i3(.e1(x[5]),.e2(y[5]),.c(c2),.s(ss[5]),.p(p3),.g(g3));
    kairo_a i4(.e1(x[6]),.e2(y[6]),.c(c3),.s(ss[6]),.p(p4),.g(g4));
    kairo_a i5(.e1(x[7]),.e2(y[7]),.c(c4),.s(ss[7]),.p(p5),.g(g5));
    kairo_a i6(.e1(x[8]),.e2(y[8]),.c(c5),.s(ss[8]),.p(p6),.g(g6));
    kairo_a i7(.e1(x[9]),.e2(y[9]),.c(c6),.s(ss[9]),.p(p7),.g(g7));
```

```
    kairo_a i8(.e1(x[10]),.e2(y[10]),.c(c7),.s(ss[10]),.p(p8),.g(g8));
    kairo_a i9(.e1(x[11]),.e2(y[11]),.c(c8),.s(ss[11]),.p(p9),.g(g9));
    kairo_a i10(.e1(x[12]),.e2(y[12]),.c(c9),.s(ss[12]),.p(p10),.g(g10));
    kairo_a i11(.e1(x[13]),.e2(y[13]),.c(c10),.s(ss[13]),.p(p11),.g(g11));
    kairo_a i12(.e1(x[14]),.e2(y[14]),.c(c11),.s(ss[14]),.p(p12),.g(g12));
    kairo_a i13(.e1(x[15]),.e2(y[15]),.c(c12),.s(ss[15]),.p(p13),.g(g13));
    kairo_a i14(.e1(x[16]),.e2(y[16]),.c(c13),.s(ss[16]),.p(p14),.g(g14));
    kairo_a i15(.e1(x[17]),.e2(y[17]),.c(c14),.s(ss[17]),.p(p15),.g(g15));
    kairo_a i16(.e1(x[18]),.e2(y[18]),.c(c15),.s(ss[18]),.p(p16),.g(g16));
    kairo_a i17(.e1(x[19]),.e2(y[19]),.c(c16),.s(ss[19]),.p(p17),.g(g17));
    kairo_a i18(.e1(x[20]),.e2(y[20]),.c(c17),.s(ss[20]),.p(p18),.g(g18));
    kairo_a i19(.e1(x[21]),.e2(y[21]),.c(c18),.s(ss[21]),.p(p19),.g(g19));
    kairo_a i20(.e1(x[22]),.e2(y[22]),.c(c19),.s(ss[22]),.p(p20),.g(g20));
    kairo_a i21(.e1(x[23]),.e2(y[23]),.c(c20),.s(ss[23]),.p(p21),.g(g21));
    kairo_a i22(.e1(x[24]),.e2(y[24]),.c(c21),.s(ss[24]),.p(p22),.g(g22));
    kairo_a i23(.e1(x[25]),.e2(y[25]),.c(c22),.s(ss[25]),.p(p23),.g(g23));

        assign ss[26] = g23  | (c22&p23);

kairo_b
i24(.p({p3,p2,p1,p0}),.g({g3,g2,g1,g0}),.c(c23),.pp(p24),.gg(g24),.cc({c2,c1,c0,c_1}));
kairo_b
i25(.p({p7,p6,p5,p4}),.g({g7,g6,g5,g4}),.c(c24),.pp(p25),.gg(g25),.cc({c6,c5,c4,c3}));
kairo_b
i26(.p({p11,p10,p9,p8}),.g({g11,g10,g9,g8}),.c(c25),.pp(p26),.gg(g26),.cc({c10,c9,c8,c7}));
kairo_b
i27(.p({p15,p14,p13,p12}),.g({g15,g14,g13,g12}),.c(c26),.pp(p27),.gg(g27),.cc({c14,c13,c12,c
11}));
kairo_b
i28(.p({p19,p18,p17,p16}),.g({g19,g18,g17,g16}),.c(c27),.pp(p28),.gg(g28),.cc({c18,c17,c16,c
15}));
kairo_b
i29(.p({p23,p22,p21,p20}),.g({g23,g22,g21,g20}),.c(c28),.pp(p29),.gg(g29),.cc({c22,c21,c20,c
19}));

    kairo_c i30(.c(c29),.p({p25,p24}),.g({g25,g24}),.cc({c24,c23}),.pp(p30),.gg(g30));
    kairo_c i31(.c(c30),.p({p27,p26}),.g({g27,g26}),.cc({c26,c25}),.pp(p31),.gg(g31));
```

```
    kairo_c i32(.c(c31),.p({p29,p28}),.g({g29,g28}),.cc({c28,c27}),.pp(p32),.gg(g32));
    kairo_c i33(.c(c32),.p({p31,p30}),.g({g31,g30}),.cc({c30,c29}),.pp(p33),.gg(g33));
    kairo_c i34(.c(c),.p({p32,p33}),.g({g32,g33}),.cc({c31,c32}),.pp(p34),.gg(g34));

  always @(ss)begin                    //丸め処理
    if({ss[1],ss[0]}>=2'b10)begin
       c<=1'b1;
    end else begin
       c<=1'b0;
    end
  end

  always @(ss)begin
       s = ss[26:2];
  end
endmodule
```

/*桁上げ先見回路を用いたsub回路（下位モジュール）*/
入力xに対してその２の補数値ｙ１を求め、後の処理はaddと同じなので省略。

```
/*回路A（下位モジュール）*/
module kairo_a(e1,e2,c,s,p,g);
    input     e1,e2,c;
    output    s,p,g;

    assign s = (e1^e2)^c;
    assign p = e1  |e2;
    assign g = e1 & e2;

endmodule
```

生成gと伝播p

```
/*回路B（下位モジュール）*/
module kairo_b(p,g,c,pp,gg,cc);
  input  [3:0] p,g;
```

４ビット単位の加算器に対応する生成ggと伝播ppと桁上げcc

```
  input         c;
  output        pp,gg;
  output [3:0] cc;

  assign pp = p[0] & p[1] & p[2] & p[3];
  assign gg = g[3]  | (p[3]&g[2])  | (p[3]&p[2]&g[1])  | (p[3]&p[2]&p[2]&g[0]);
  assign cc[0] = c;
  assign cc[1] = g[0]  | (cc[0]&p[0]);
  assign cc[2] = g[1]  | (cc[1]&p[1]);
  assign cc[3] = g[2]  | (cc[2]&p[2]);

endmodule
```

2つの 4 ビット単位の加算器に対応する生成ggと伝播ppと桁上げcc[2]

```
/*回路C（下位モジュール）*/
module kairo_c(c,p,g,cc,pp,gg);
  input          c;
  input  [1:0]   p,g;
  output [1:0]   cc;
  output         pp,gg;

  assign cc[1] = g[0]  | (c&p[0]);
  assign cc[0] = c;
  assign pp = p[0] & p[1];
  assign gg = g[1]  | (p[1]&g[0]);

endmodule
```

順次桁上げ加算器
（丸め処理）

```
/*回路D（下位モジュール）*/
module kairo_d(x,y,c,cc,s);
   input   x,y,c;
   output  s,cc;

   assign cc = (x&y)  | (x&c)  |(y&c);
   assign s = (x^y)^c;
endmodul
```

## １．２　加減算器のシミュレーションによるバグだし

１．１で説明した加減算器のシミュレーションを行い、計算できないところを探す。ここでは、テストベンチ記述を説明した後、シミュレーションを実行し、その結果を考察する。

### １．２．１　テストベンチの記述

入力の正負や大小を考慮した、３０回の演算を行う。

```
//* 加減算器のテストベンチ記述 *//
module fau_test;                                    /*テストベンチ名*/
      reg  [31:0]   a,b;                           /*検証対象への入力はreg宣言*/
      reg           FAUsel;
      wire [31:0]   out;                           /*検証対象の出力はwire宣言*/

      fau A(a,b,FAUsel,out);                       /*検証対象の記述*/

/*テスト入力の記述*/
      initial begin
               a=32'h40a00000; b=32'h00000000; FAUsel=1'b0;     //(5)+(0)=
         #1000 a=32'h00000000; b=32'h40a00000; FAUsel=1'b0;     //(0)+(5)=
         #1000 a=32'hc0e00000; b=32'h00000000; FAUsel=1'b0;     //(-7)+(0)=
         #1000 a=32'h00000000; b=32'hc0e00000; FAUsel=1'b0;     //(0)+(-7)=
         #1000 a=32'h00000000; b=32'h00000000; FAUsel=1'b0;     //(0)+(0)=
         #1000 a=32'h40a00000; b=32'hc0e00000; FAUsel=1'b0;     //(5)+(-7)=
         #1000 a=32'h40e00000; b=32'hc0a00000; FAUsel=1'b0;     //(7)+(-5)=
         #1000 a=32'h40e00000; b=32'hc0e00000; FAUsel=1'b0;     //(7)+(-7)=
         #1000 a=32'hc0a00000; b=32'h40e00000; FAUsel=1'b0;     //(-5)+(7)=
         #1000 a=32'hc0e00000; b=32'h40a00000; FAUsel=1'b0;     //(-7)+(5)=
         #1000 a=32'hc0a00000; b=32'h40a00000; FAUsel=1'b0;     //(-5)+(5)=
         #1000 a=32'hc0a00000; b=32'hc0e00000; FAUsel=1'b0;     //(-5)+(-7)=
         #1000 a=32'h40a00000; b=32'h40e00000; FAUsel=1'b0;     //(5)+(-7)=
         #1000 a=32'h40a00000; b=32'h40a00000; FAUsel=1'b0;     //(5)+(5)=
         #1000 a=32'h40e00000; b=32'h40e00000; FAUsel=1'b0;     //(7)+(7)=

         #1000 a=32'h40a00000; b=32'h00000000; FAUsel=1'b1;     //(5)-(0)=
         #1000 a=32'h00000000; b=32'h40a00000; FAUsel=1'b1;     //(0)-(5)=
         #1000 a=32'hc0e00000; b=32'h00000000; FAUsel=1'b1;     //(-7)-(0)=
         #1000 a=32'h00000000; b=32'hc0e00000; FAUsel=1'b1;     //(0)-(-7)=
         #1000 a=32'h00000000; b=32'h00000000; FAUsel=1'b1;     //(0)-(0)=
         #1000 a=32'h40a00000; b=32'hc0e00000; FAUsel=1'b1;     //(5)-(-7)=
         #1000 a=32'h40e00000; b=32'hc0a00000; FAUsel=1'b1;     //(7)-(-5)=
         #1000 a=32'h40e00000; b=32'hc0e00000; FAUsel=1'b1;     //(7)-(-7)=
         #1000 a=32'hc0a00000; b=32'h40e00000; FAUsel=1'b1;     //(-5)-(7)=
         #1000 a=32'hc0e00000; b=32'h40a00000; FAUsel=1'b1;     //(-7)-(5)=
```

```
        #1000    a=32'hc0a00000; b=32'h40a00000; FAUsel=1'b1;      //( -5) -(5)=
        #1000    a=32'hc0a00000; b=32'hc0e00000; FAUsel=1'b1;      //( -5) -( -7)=
        #1000    a=32'h40a00000; b=32'h40e00000; FAUsel=1'b1;      //(5) -( -7)=
        #1000    a=32'h40a00000; b=32'h40a00000; FAUsel=1'b1;      //(5) -(5)=
        #1000    a=32'h40e00000; b=32'h40e00000; FAUsel=1'b1;      //(7) -(7)=
        #1000    $finish;
    end

endmodule
```

### 1．2．2　シミュレーションの実行

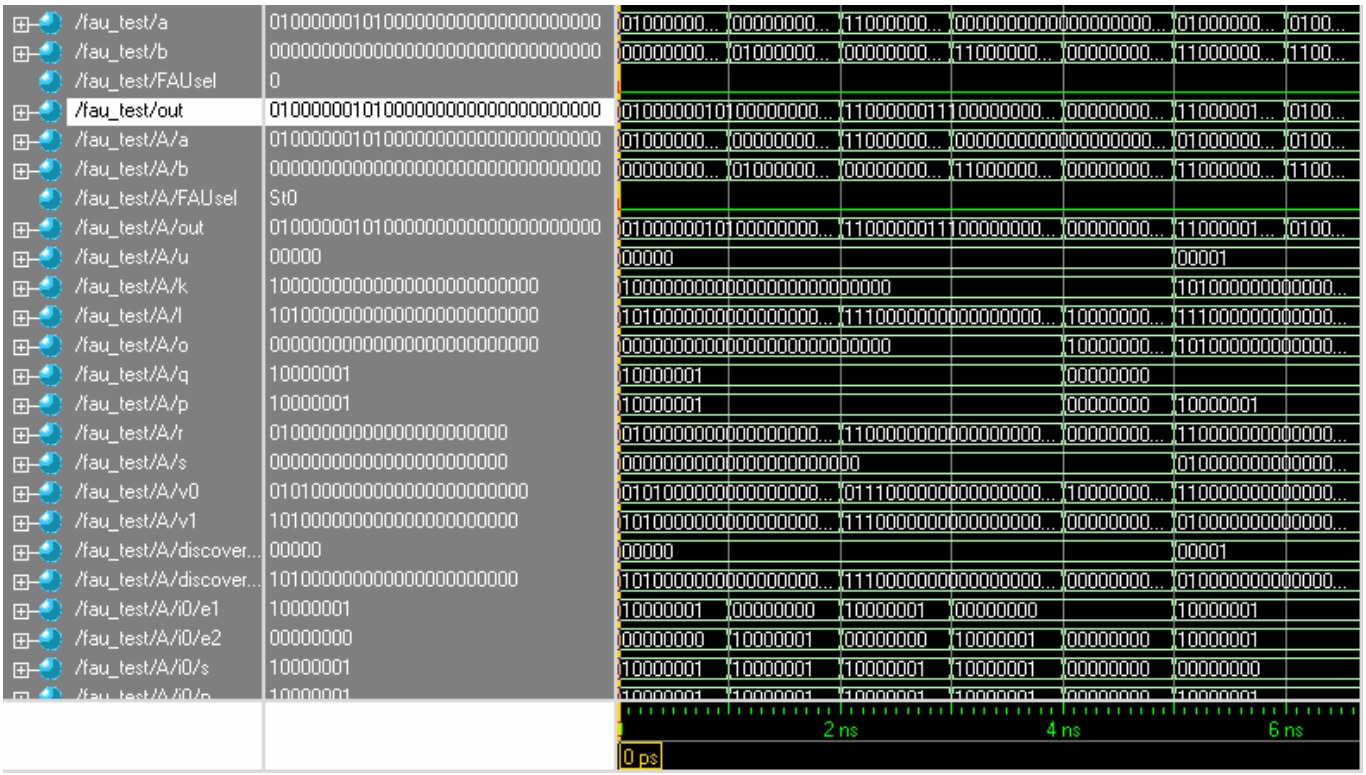


図1．5　加減算器のシミュレーションの波形表示（一部省略）

### 1．2．3　シミュレーション結果の説明

表1．3を見ると、ほとんどの計算が間違っている。この結果から、0＋0、正＋負、負＋正と、ほとんどの減算で正しい答えが出てきていない。

| (ns) | 入力a | 入力b | 加算･減算 | 出力out | 正しい答え | 結果 |
|---|---|---|---|---|---|---|
| 0 | 5 | 0 | 加算 | 5 | 5 | ○ |
| 1 | 0 | 5 | 加算 | 5 | 5 | ○ |
| 2 | -7 | 0 | 加算 | -7 | -7 | ○ |
| 3 | 0 | -7 | 加算 | -7 | -7 | ○ |
| 4 | 0 | 0 | 加算 | $2^{-126}$ | 0 | × |
| 5 | 5 | -7 | 加算 | -12 | -2 | × |
| 6 | 7 | -5 | 加算 | 12 | 2 | × |
| 7 | 7 | -7 | 加算 | 14 | 0 | × |
| 8 | -5 | 7 | 加算 | 12 | 2 | × |
| 9 | -7 | 5 | 加算 | -12 | -2 | × |
| 10 | -5 | 5 | 加算 | -10 | 0 | × |
| 11 | -5 | -7 | 加算 | -12 | -12 | ○ |
| 12 | 5 | 7 | 加算 | 12 | 12 | ○ |
| 13 | 5 | 5 | 加算 | 10 | 10 | ○ |
| 14 | 7 | 7 | 加算 | 14 | 14 | ○ |
| 15 | 5 | 0 | 減算 | 5 | 5 | ○ |
| 16 | 0 | 5 | 減算 | 5 | -5 | × |
| 17 | -7 | 0 | 減算 | -7 | -7 | ○ |
| 18 | 0 | -7 | 減算 | -7 | 7 | × |
| 19 | 0 | 0 | 減算 | 0 | 0 | ○ |
| 20 | 5 | -7 | 減算 | -2 | 12 | × |
| 21 | 7 | -5 | 減算 | 2 | 12 | × |
| 22 | 7 | -7 | 減算 | 2 | 14 | × |
| 23 | -5 | 7 | 減算 | 2 | -12 | × |
| 24 | -7 | 5 | 減算 | -2 | -12 | × |
| 25 | -5 | 5 | 減算 | -2 | -10 | × |
| 26 | -5 | -7 | 減算 | -2 | 2 | × |
| 27 | 5 | 7 | 減算 | 2 | -2 | × |
| 28 | 5 | 5 | 減算 | 2 | 0 | × |
| 29 | 7 | 7 | 減算 | 2 | 0 | × |

表1．3　シミュレーション結果（加減算器）

## １．３　加減算器の機能検証と新加減算器の設計・シミュレーション

１．２で、加減算器の正しい答えの出ない部分がわかった。ここでは、正しい答えが出るように加減算器を修正する。

### １．３．１　加減算器の機能検証

図１．６を見ると、大小判別のブロックから大きい方の符号が全体の出力の符号部 out[31]にそのまま出力されている。つまり、入力 a と入力ｂの仮数部と指数部から判定された、a,b の絶対値比較により大きいと判定された方の符号がそのまま out[31]に出力されているのである。これは、間違いである。例えば、５－７の演算をする時、答えは－２とならなければならない。しかし、この加減算器で計算すると、出力される符号は５と７を絶対値比較して大きい方、つまり７の方の符号が out[31]に出力されてしまうので、答えは＋２となってしまう。

また、この加減算器には、add と sub の二つの加算器（ALU）がある。これは、２の補数での演算をすることで、一つにすることができる。

正規化のところで、FAUsel＝0（加算）の時は v0（add の出力）を、FAUsel＝1（減算）の時は v1（sub の出力）を out に出力するようになっている。例えば、5－(－7)の計算をする時、答えは 12 とならなければいけないのに、FAUsel＝1 なので out[23:0]には、sub の結果 2 が出力されてしまうので、間違っていることがわかる。

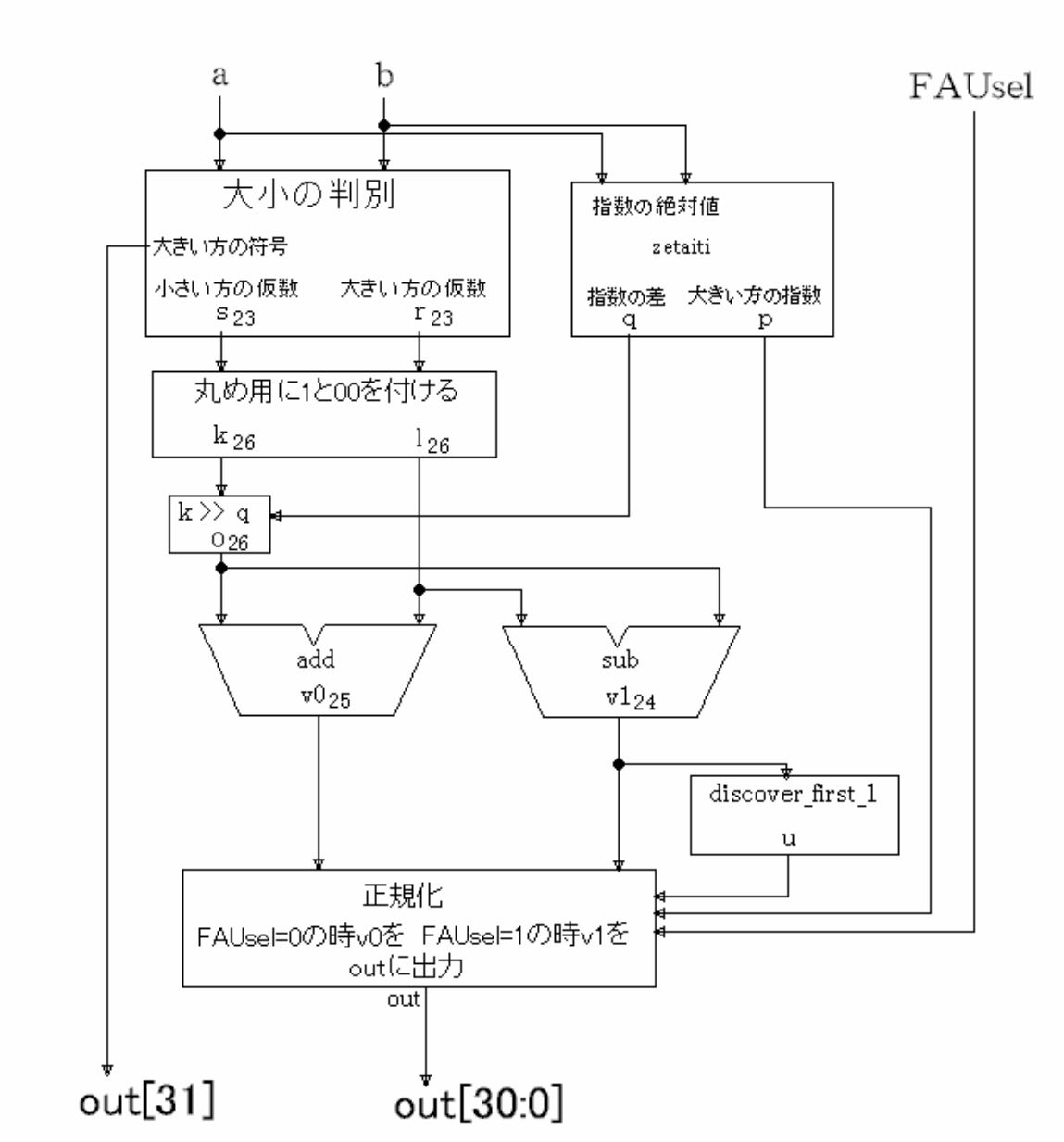


**図1．6　加減算器のブロック図**

### １．３．２　新加減算器の設計

（1）　新加減算器のブロック図

１．３．１の検証を元に、新しい加減算器を設計していく。新加減算器には、次のような要素を新しく追加した。

・　まず、FAUsel でｂをｂ１に変換する。
　　これは、例えば(５)－(７)のような減算は、(５)+(－７)のような加算に置き換える。
・　大小の判別のブロックに、「小さい方の符号：S」と「大きい方の符号：R」の出力を追加した。
　　２の補数に変換する時、最後の正規化の時に使われる。
・　加算器を１つにした。
　　旧加減算器にあった sub を消去し、２の補数に変換して計算することで、加算器を add だけにした。

このようにして設計しなおした新加減算器のブロック図を図１．７に示す。

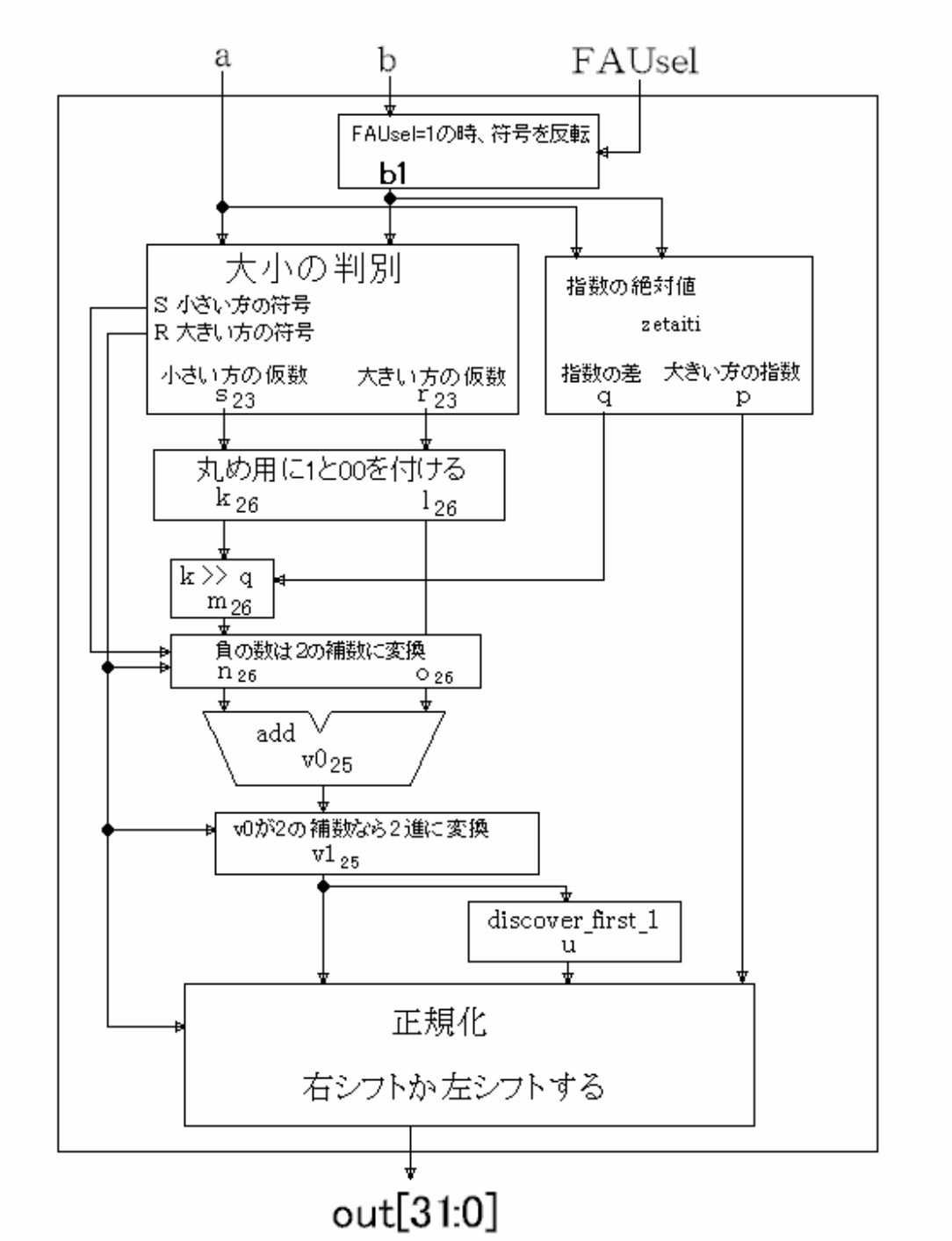


**図1．7　新加減算器のブロック図**

（2）新加減算器の HDL 記述

次に図１．７の新加減算器のブロック図を元に、旧加減算器の HDL 記述を修正し、新加減算器を VerilogHDL で記述する。なお、下位モジュールは変更していないので、省略する。

```
/*新加減算器の HDL 記述（上位モジュール）*/
module fau(a,b,FAUsel,out);
    input  [31:0]  a,b;
    input          FAUsel;
    output [31:0]  out;
    reg    [31:0]  out;
    wire   [31:0]  b1;
    wire   [7:0]   u;
    wire   [25:0]  k,l,m;
    reg    [25:0]  n,o;
    wire   [7:0]   q;
    wire   [7:0]   p;
    reg    [30:0]  r,s;
    wire   [24:0]  v0;
    reg    [24:0]  v1;
    reg            R,S;          //大きい数値の符号(R)と小さい数値の符号(S)

  assign b1 = (FAUsel==1'b1) ? {~b[31],b[30:0]} : b ;     //FAUsel=1 の時、符号ビットを反転

//絶対値にした２つの数値の大小比較//
  always @(a or b1)begin
    if(a[30:23]>b1[30:23])begin              //指数部で大小を比較
        s <= b1[30:0];                        //←小さな数値の指数部・仮数部（s）
        r <= a[30:0];                         //←大きな数値の指数部・仮数部（r）
        S <= b1[31];                          //←小さい数値の符号部（S）
        R <= a[31];                           //←大きい数値の符号部（R）
    end else if(a[30:23]<b1[30:23])begin
        s <= a[30:0];
        r <= b1[30:0];
        S <= a[31];
        R <= b1[31];
    end else if(a[30:23]==b1[30:23])begin    //指数部が等しいとき
        if(a[22:0]>b1[22:0])begin
              s <= b1[30:0];
              r <= a[30:0];
              S <= b1[31];
              R <= a[31];
        end else if(a[22:0]<b1[22:0])begin
```

```
                s <= a[30:0];
                r <= b1[30:0];
                S <= a[31];
                R <= b1[31];
            end else if(a[22:0]==b1[22:0])begin
                if(a[31]==1'b1 && b1[31]==1'b0) begin
                    s <= a[30:0];
                    r <= b1[30:0];
                    S <= a[31];
                    R <= b1[31];
                end else begin
                s <= b1[30:0];
                r <= a[30:0];
                S <= b1[31];
                R <= a[31];
                end
            end
        end
    end

//仮数部と数値部がすべて0のとき符号部も0にそろえる//
    always @(s or r) begin
        if(s==32'h00000000) begin
            S <= 1'b0;
        end
        if(r==32'h00000000) begin
            R <= 1'b0;
        end
    end

    zetaiti i0(.e1(a[30:23]),.e2(b[30:23]),.s(q),.p(p));  //指数の差（q）
                                                          //大きな数値の指数部（p）

 //sとrの先頭に1または0を,後ろに丸め用の00を付加//
    assign k = ( s==1'b0) ? {1'b0,s[22:0],2'b00} : {1'b1,s[22:0],2'b00};
    assign l = ( r==1'b0) ? {1'b0,r[22:0],2'b00} : {1'b1,r[22:0],2'b00};

    assign m = k >> q;                                    //小さな数値を指数の差だけ右にシフト

    always @(R or S or l or m) begin                      //負の数値を２の補数に変換
            if(R==1'b1) begin
                n <= -l + 1'b1;
            end else begin
                n <= l;
```

```
        end
        if(S==1'b1) begin
            o <= ~m + 1'b1;
        end else begin
            o <= m;
        end
end

add i1(.x(n),.y(o),.s(v0));                         //加算

always @(R or v0) begin                             //v0が補数ならば2進に直す
        if((R==1'b1) && (S==1'b1)) begin
            v1 <= ~v0[24:0] + 1'b1;
        end else if(R==1'b1) begin
            v1 <= {v0[24],(~v0[23:0] + 1'b1)};
        end else if(S==1'b1) begin
            v1 <= {1'b0,v0[23:0]};
        end else begin
            v1 <= v0;
        end
end

function [4:0] discover_first_1;
        input  [24:0]  t;
        if(t[24]==1'b1)       discover_first_1 = 5'b11111;
        else if(t[23]==1'b1)  discover_first_1 = 5'b00000;
        else if(t[22]==1'b1)  discover_first_1 = 5'b00001;
        else if(t[21]==1'b1)  discover_first_1 = 5'b00010;
        else if(t[20]==1'b1)  discover_first_1 = 5'b00011;
        else if(t[19]==1'b1)  discover_first_1 = 5'b00100;
        else if(t[18]==1'b1)  discover_first_1 = 5'b00101;
        else if(t[17]==1'b1)  discover_first_1 = 5'b00110;
        else if(t[16]==1'b1)  discover_first_1 = 5'b00111;
        else if(t[15]==1'b1)  discover_first_1 = 5'b01000;
        else if(t[14]==1'b1)  discover_first_1 = 5'b01001;
        else if(t[13]==1'b1)  discover_first_1 = 5'b01010;
        else if(t[12]==1'b1)  discover_first_1 = 5'b01011;
        else if(t[11]==1'b1)  discover_first_1 = 5'b01100;
        else if(t[10]==1'b1)  discover_first_1 = 5'b01101;
        else if(t[9]==1'b1)   discover_first_1 = 5'b01110;
        else if(t[8]==1'b1)   discover_first_1 = 5'b01111;
        else if(t[7]==1'b1)   discover_first_1 = 5'b10000;
        else if(t[6]==1'b1)   discover_first_1 = 5'b10001;
        else if(t[5]==1'b1)   discover_first_1 = 5'b10010;
```

```
            else if(t[4]==1'b1)   discover_first_1 = 5'b10011;
            else if(t[3]==1'b1)   discover_first_1 = 5'b10100;
            else if(t[2]==1'b1)   discover_first_1 = 5'b10101;
            else if(t[1]==1'b1)   discover_first_1 = 5'b10110;
            else if(t[0]==1'b1)   discover_first_1 = 5'b10111;
    endfunction

    assign u = discover_first_1(v1[24:0]);

    always @(v1 or p or u or R) begin
            if((p[7:0]==8'b00000000) || (v1==32'h00000000)) begin
              out[31:0] <= 32'h00000000;
            end else begin
                if(u==5'b11111)begin                //正規化(右シフト)
                    out[22:0] <= v1 >> 1'b1;
                    out[30:23] <= p + 1'b1;
                    out[31] <= R;
                end else begin                      //正規化(左シフト)
                    out[22:0] <= v1 << u;
                    out[30:23] <= p  - u;
                    out[31] <= R;
                end
            end
    end
endmodule
```

### １．３．３　新加減算器のシミュレーション

１．３．２で記述した新加減算器のシミュレーションを行い、正しく計算できるか確認する。

#### １．３．３．１　テストベンチの記述

ここでは、旧加減算器とおなじものを使用する。

```
//* 新加減算器のテストベンチ記述 *//
module fau_test;                                    /*テストベンチ名*/
      reg   [31:0]    a,b;                          /*検証対象への入力はreg宣言*/
      reg             FAUsel;
      wire  [31:0]    out;                          /*検証対象の出力はwire宣言*/

      fau A(a,b,FAUsel,out);                        /*検証対象の記述*/

/*テスト入力の記述*/
      initial begin
                a=32'h40a00000; b=32'h00000000; FAUsel=1'b0;      //(5)+(0)=
         #1000  a=32'h00000000; b=32'h40a00000; FAUsel=1'b0;      //(0)+(5)=
         #1000  a=32'hc0e00000; b=32'h00000000; FAUsel=1'b0;      //(-7)+(0)=
         #1000  a=32'h00000000; b=32'hc0e00000; FAUsel=1'b0;      //(0)+(-7)=
         #1000  a=32'h00000000; b=32'h00000000; FAUsel=1'b0;      //(0)+(0)=
         #1000  a=32'h40a00000; b=32'hc0e00000; FAUsel=1'b0;      //(5)+(-7)=
         #1000  a=32'h40e00000; b=32'hc0a00000; FAUsel=1'b0;      //(7)+(-5)=
         #1000  a=32'h40e00000; b=32'hc0e00000; FAUsel=1'b0;      //(7)+(-7)=
         #1000  a=32'hc0a00000; b=32'h40e00000; FAUsel=1'b0;      //(-5)+(7)=
         #1000  a=32'hc0e00000; b=32'h40a00000; FAUsel=1'b0;      //(-7)+(5)=
         #1000  a=32'hc0a00000; b=32'h40a00000; FAUsel=1'b0;      //(-5)+(5)=
         #1000  a=32'hc0a00000; b=32'hc0e00000; FAUsel=1'b0;      //(-5)+(-7)=
         #1000  a=32'h40a00000; b=32'h40e00000; FAUsel=1'b0;      //(5)+(-7)=
         #1000  a=32'h40a00000; b=32'h40a00000; FAUsel=1'b0;      //(5)+(5)=
         #1000  a=32'h40e00000; b=32'h40e00000; FAUsel=1'b0;      //(7)+(7)=

         #1000  a=32'h40a00000; b=32'h00000000; FAUsel=1'b1;      //(5)-(0)=
         #1000  a=32'h00000000; b=32'h40a00000; FAUsel=1'b1;      //(0)-(5)=
         #1000  a=32'hc0e00000; b=32'h00000000; FAUsel=1'b1;      //(-7)-(0)=
         #1000  a=32'h00000000; b=32'hc0e00000; FAUsel=1'b1;      //(0)-(-7)=
         #1000  a=32'h00000000; b=32'h00000000; FAUsel=1'b1;      //(0)-(0)=
         #1000  a=32'h40a00000; b=32'hc0e00000; FAUsel=1'b1;      //(5)-(-7)=
         #1000  a=32'h40e00000; b=32'hc0a00000; FAUsel=1'b1;      //(7)-(-5)=
         #1000  a=32'h40e00000; b=32'hc0e00000; FAUsel=1'b1;      //(7)-(-7)=
         #1000  a=32'hc0a00000; b=32'h40e00000; FAUsel=1'b1;      //(-5)-(7)=
         #1000  a=32'hc0e00000; b=32'h40a00000; FAUsel=1'b1;      //(-7)-(5)=
```

```
        #1000    a=32'hc0a00000; b=32'h40a00000; FAUsel=1'b1;      //(-5)-(5)=
        #1000    a=32'hc0a00000; b=32'hc0e00000; FAUsel=1'b1;      //(-5)-(-7)=
        #1000    a=32'h40a00000; b=32'h40e00000; FAUsel=1'b1;      //(5)-(-7)=
        #1000    a=32'h40a00000; b=32'h40a00000; FAUsel=1'b1;      //(5)-(5)=
        #1000    a=32'h40e00000; b=32'h40e00000; FAUsel=1'b1;      //(7)-(7)=
        #1000    $finish;
    end

endmodule
```

### 1．3．3．2　シミュレーションの実行

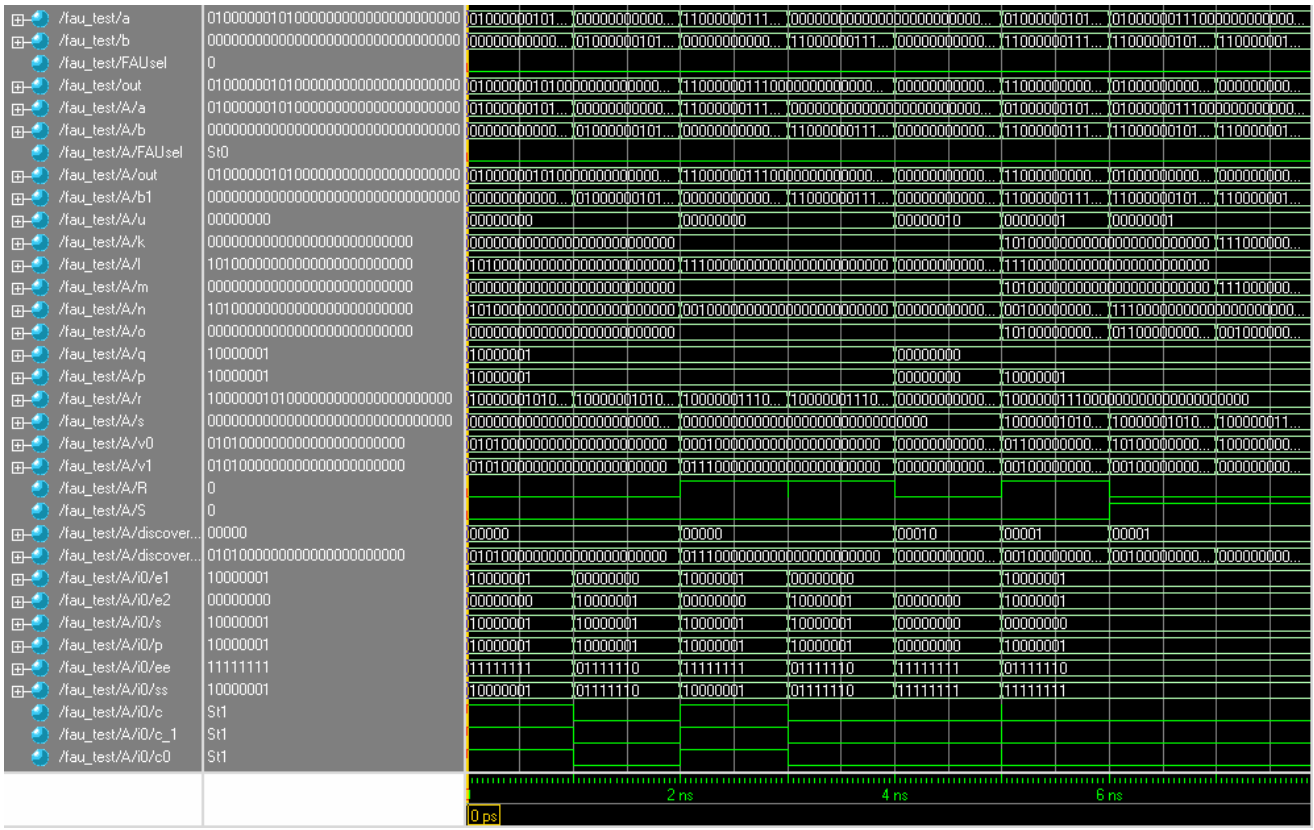


図1．8　新加減算器シミュレーションの波形表示（一部省略）

### 1．3．3．3　シミュレーション結果

図1．9を見てわかるように、すべての演算で正しい解が出た。よって、新加減算器は正しい演算ができることが分かる。

| (ns) | 入力a | 入力b | 加算･減算 | 出力out | 正しい答え | 結果 |
|---|---|---|---|---|---|---|
| 0 | 5 | 0 | 加算 | 5 | 5 | ○ |
| 1 | 0 | 5 | 加算 | 5 | 5 | ○ |
| 2 | -7 | 0 | 加算 | -7 | -7 | ○ |
| 3 | 0 | -7 | 加算 | -7 | -7 | ○ |
| 4 | 0 | 0 | 加算 | 0 | 0 | ○ |
| 5 | 5 | -7 | 加算 | -2 | -2 | ○ |
| 6 | 7 | -5 | 加算 | 2 | 2 | ○ |
| 7 | 7 | -7 | 加算 | 0 | 0 | ○ |
| 8 | -5 | 7 | 加算 | 2 | 2 | ○ |
| 9 | -7 | 5 | 加算 | -2 | -2 | ○ |
| 10 | -5 | 5 | 加算 | 0 | 0 | ○ |
| 11 | -5 | -7 | 加算 | -12 | -12 | ○ |
| 12 | 5 | 7 | 加算 | 12 | 12 | ○ |
| 13 | 5 | 5 | 加算 | 10 | 10 | ○ |
| 14 | 7 | 7 | 加算 | 14 | 14 | ○ |
| 15 | 5 | 0 | 減算 | 5 | 5 | ○ |
| 16 | 0 | 5 | 減算 | -5 | -5 | ○ |
| 17 | -7 | 0 | 減算 | -7 | -7 | ○ |
| 18 | 0 | -7 | 減算 | 7 | 7 | ○ |
| 19 | 0 | 0 | 減算 | 0 | 0 | ○ |
| 20 | 5 | -7 | 減算 | 12 | 12 | ○ |
| 21 | 7 | -5 | 減算 | 12 | 12 | ○ |
| 22 | 7 | -7 | 減算 | 14 | 14 | ○ |
| 23 | -5 | 7 | 減算 | -12 | -12 | ○ |
| 24 | -7 | 5 | 減算 | -12 | -12 | ○ |
| 25 | -5 | 5 | 減算 | -10 | -10 | ○ |
| 26 | -5 | -7 | 減算 | 2 | 2 | ○ |
| 27 | 5 | 7 | 減算 | -2 | -2 | ○ |
| 28 | 5 | 5 | 減算 | 0 | 0 | ○ |
| 29 | 7 | 7 | 減算 | 0 | 0 | ○ |

図1．9　シミュレーション結果（新加減算器）

## ２　メモリの回路設計

メモリ LSI は“情報（データ）を記憶する”LSI で、その応用範囲は広く、スーパーコンピュータから家庭電化製品まで幅広く使われ、我々の生活を支えている。例えば、DRAM では、四半世紀で 100 万倍も大容量化されている。これは３年間４倍の割合で増加したことになる。

### ２．１　メモリ LSI の分類

メモリＬＳＩは、用途、記憶セルの構造、使用するトランジスタの種類などにより分類される。用途による分類では、表２．１に示すように、汎用(標準)メモリおよび専用（特定用途向け）メモリに分けられる。汎用メモリはさらに記憶セルの構造により、ランダムアクセスメモリ（Random Access Memory；RAM）と、リードオンリーメモリ（Read Only Memory；ROM）に大別できる。

RAMは、任意の番地のメモリセルに自由にアクセスして、随時、データを読み出したり書き込んだりでる。また読み書きを同じ速さで実行できるメモリで、メモリセルの構造と特性の違いからダイナミック RAM（DRAM）とスタティック RAM（SRAM）に分類される。

<table>
<tr><td rowspan="6">汎用<br>(標準)</td><td>ランダムアクセスメモリ（RAM）</td><td colspan="3">ダイナミック RAM（DRAM）</td></tr>
<tr><td></td><td colspan="3">スタティック RAM（SRAM）</td></tr>
<tr><td rowspan="4">リードオンリーメモリ<br>（ROM）</td><td colspan="3">マスク ROM</td></tr>
<tr><td rowspan="3">プログラム ROM<br>（PROM）</td><td rowspan="3">書き換え可能</td><td>EPROM</td></tr>
<tr><td>EEPROM</td></tr>
<tr><td>フラッシュメモリ</td></tr>
<tr><td colspan="2">書き換え不可能</td><td>フューズ ROM</td></tr>
<tr><td>専用<br>（特定用途向け）</td><td colspan="4">画像メモリ<br>シンクロナスメモリ（SDRAM）<br>キャッシュメモリ</td></tr>
</table>

表２．１　記憶セルの構造によるメモリ LSI の分類

ここからは、代表的な LSI メモリ RAM について詳しく説明していく。

### ２．２　メモリ LSI の基本構成と動作

#### ２．２．１　基本構成

代表的な LSI メモリである RAM の基本構成はメモリセルアレイ、周辺回路および入出力インターフェース回路から構成される。入力信号としてビットアドレス信号、クロック信号、書き込み入力信号、

書き込み制御信号があり、出力信号として、読み出し出力信号がある。

① メモリセルアレイは通常1ビットの情報を記憶するメモリセルがマトリックス状に$2^{N\times M}$個配置されたアレイ部、$2^{M}$個のセルが接続されたワード線$2^{N}$本、$2^{N}$個のセルが接続されたデータ対線$2^{M}$対からなる。
② 入出力インターフェース回路には行アドレス、列アドレス、クロック信号、書き込み信号、読み出し信号などをラッチする各種のバッファがある。
③ 周辺回路はアレイを制御する直接周辺回路（デコーダ、ドライバ、マルチプレクサ、書き込み回路、読み出し回路、I/O制御信号）および直接周辺回路を制御する間接周辺回路からなる。

### 2．2．2　基本動作

記憶データの読み出し操作は、まず、行、列アドレスバッファにアドレス情報がラッチされるところからはじまる。アドレス情報に基づいて読み出しのセルの行と列アドレスが決定されると、対応する行ドライバの出力が高レベルとなる。この結果、選択されたワード線が活性化され、これに接続されるすべてのメモリセルのデータがデータ対線にゆっくり現われる。
これと同時に、対応する列ドライバの出力も高レベルとなるので、選択されたデータはセンスアンプに接続され、センスアンプが活性化されると、このゆっくりしたデータ対線の信号変化は、センスアンプにより急速に増加され、読み出し回路を介して出力される。

書き込み動作も特定のメモリセルが選択されるまでは読み出し動作と同じである。メモリセルが選択され、書き込み回路が書き込み制御信号により活性化されると、入力データがメモリセルに書き込まれる。これは記憶されてたデータの上に上書きするかのように、新たなデータが強制的に置き換えられる。

## 2．3　DRAMの基本構造

DRAM（Dynamic Random Access Memory）は中速ながら記憶密度が高く、価格も安い汎用メモリである。その特徴から、大容量メモリを必要とする汎用コンピュータの主メモリ、低価格に重点を置くPCの主メモリなどに広く応用されてて、DRAMはメモリLSI市場全体の2／3を占め、最も代表的な標準メモリと言われている。

### 1．メモリセル

DRAMメモリセルは、図2.1に示すように電荷を蓄えるためのキャパシタ$C_1$と、$C_1$への電荷の充放電を制御するnMOSFETスイッチの2素子からなる。このようにメモリセル構成が大変簡単なので、個々のメモリセル面積をきわめて小さくできる。このため記憶密度を極めて高めることができる。
記憶内容“1”“0”は$C_1$電荷があるか否か（すなわちノードNの電圧が高いか低いか）に対応させ

る。

**2．データの書き込み**

**ワード線を高電位にしてnMOSFETスイッチを導通させ、データ線から書き込みデータ“１”、“０”に応じてノードNを高電位あるいは低電位に設定することにより実行される。**

**3．データの読み出し**

**ワード線を高電位にしてnMOSFETスイッチを導通させ、次にC1の電荷の有無をデータ線電位の高低に対応させる。**

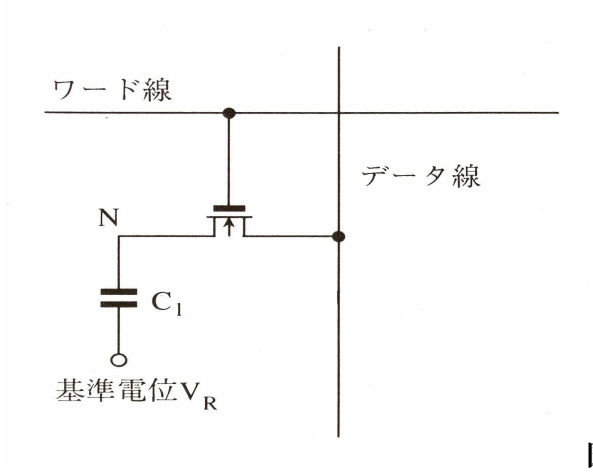


**図2．1　DRAMのメモリセル**

**メモリセルのキャパシタに電荷があるかないかで、セルノード（N）の電位の高低に対応させ、二進情報を記憶している。しかし、nMOSFETスイッチのpn接合部に流れる漏洩電流によって、キャパシタに蓄積された電荷は短時間のうちに消失し、記憶情報が破壊される。この消失を防ぐために、各ワード線に対して、一定期間ごとにデータをD線あるいはD̄線に読み出し、センスアンプで増幅して、これを再書き込みし、初期の値に再生して、データを確保するというDRAM固有の“リフレッシュ動作”がある。**

## ２．４　SRAM の基本構造

SRAM（Static Random Access Memory）のメモリセルは、図２．２に示すように、データを記憶するフリップフロップと２個のスイッチ nMOSFET からなる。SRAM は集積度で DRAM に劣るが、以下の優れた特徴がある。

（１）読み出し/書き込み動作が極めて速い
（理由：メモリセル‐データ対線―読み出し書き込み回路を介しデータのやり取りを相補信号を使って行うため）

（２）読み出し/書き込み動作のサイクルタイムをアクセスタイムに近づけることができる
（理由：リフレッシュ動作が不要なため）
サイクルタイム・・・読み出しあるいは書き込み動作がはじまってから、次の動作読み出しあるいは書き込み動作がはじまるまでの時間を言う。
アクセスタイム・・・アドレス情報がラッチされ、読み出しデータが出てくるまでの時間を言う。

（３）待機時の消費電力がきわめて小さい
（理由：メモリセルに貫通電流が流れない、リフレッシュ動作が不要なため）

（４）システムへの組み込みが大変容易
（理由：行・列アドレス信号が同時に入力され、リフレッシュ動作が不要、制御回路が小さい、作動タイミングのとり方が簡単などのため）

### １．メモリセル

SRAM のメモリセルは２個のインバータからなるフリップフロップ、ワード線で駆動される２個の nMOSFET スイッチ、データ対線より構成される。“１”と“０”の記憶はフリップフロップの双安定状態を利用している。SRAM は DRAM と異なり、データがリーク電流によって消滅することはない。したがって SRAM は DRAM で用いたリフレッシュ動作を必要としない。

### ２．データの書き込み

いま、記憶データが“１”（N が“１”レベル、Ñ が“０”レベル）とする。これを“０”（前者とは逆）に書き換える。ワード線を高電圧にして２つの nMOSFET スイッチを導通させる。書き込みデータ(D＝0)が D 線、Ð 線を介して、メモリセルへ供給される。この結果、元の記憶データは強制的に書き換えられる。書き込み制御が起動する前は、N、Ñ の電位はそれぞれ３．３V、０V であるが、起動すると N、Ñ はそれぞれ０V、３．３V に逆転する。

（D が“０”のため、Ñ の上位の p MOSFET が導通して $V_D$ から３．３V が流れてきて、Ñ は“１”レベルになる。逆に Ð が“１”のため N の下位の nMOSFET が導通して N に蓄えられていた電荷が抜けて、“０”レベルになる）

3．データの読み出し

ワード線を高電位にして２つの nMOSFET スイッチを導通させ、次にNと Ñ の電荷の有無をデータ線Dと Ð の電位にそれぞれ対応させ、読み出し回路で反転増幅して出力する。

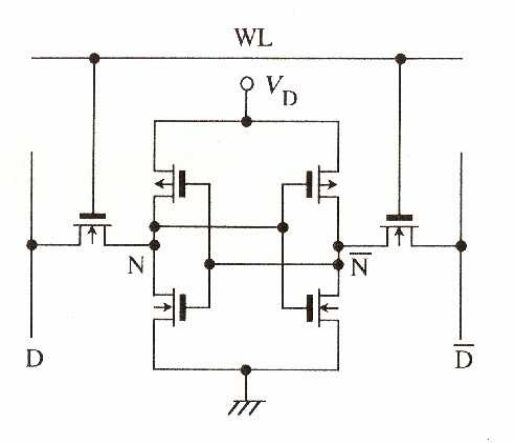


図２．２　SRAM のメモリセル

## ２．５　プリチャージ方式

データの読み出しと書き込みの前にデータ対線をある定められた電圧に必ず初期設定する。この動作をプリチャージ動作と言う。原則として、読み出しの困難を避けるために、２つのビットラインを同じ電圧にし、十分高い電圧にセットすることは必要とされる。簡単にいえば、ビットラインの電圧を等しくすることはプロセッサの $V_{DD}$ まで値を供給するプリチャージによって、一般的に成し遂げられる。

### ２．５．１　Static pullup precharg

Static pullup precharg は常に電源電圧から電圧をかけていく装置で、その利点としては、配列を横切るようなクロック信号が必要ないということ。より大きなプリチャージ装置を使うと、ビットラインの同等化が速くできる。しかしプリチャージ装置が常に ON 状態で高電圧なので、ビットラインを低くしたい時に困難になる。だから low になるレベルの大きさ考えなければならない。

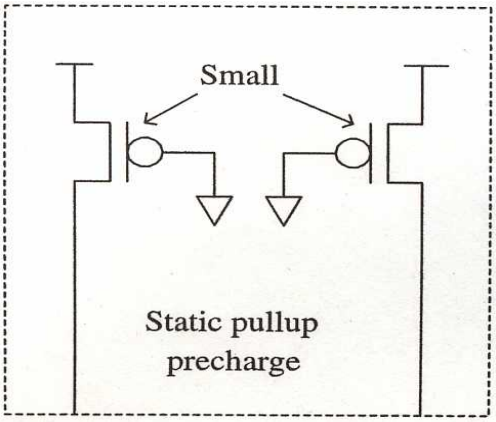

図2．3　Static pullup precharge

### 2．5．2　Clocked precharge

二つのビットラインの間につながれている３つの装置を含む回路が clocked プリチャージである。この装置は、静電容量と pullup 装置によって二つのビットラインの同等化を速くできる。しかし、クロック信号を ON にしてプリチャージするので、電力を消費する点で不利である。さらに、プリチャージクロックの正確なタイミングを作るのが難しい。

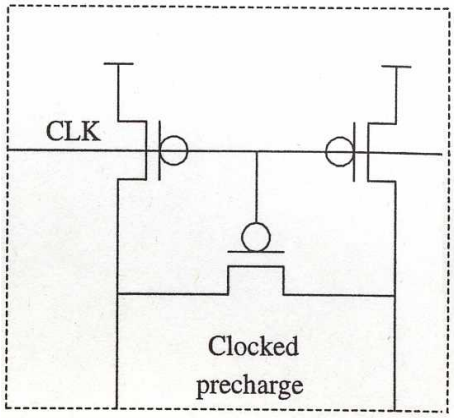

図2．4　clocked precharge

## ２．６　センスアンプ

ビットライン上の電圧を探知し、見つけた電圧差を増幅するために使われるのがセンスアンプの装置である。センスアンプの操作は CLK 信号が“ low ”の状態から始まる。P3、P4 は Sense#と Sense を $V_{DD}$ へセットする。ビットラインに影響ある N2、N4 も $V_{DD}$ にプリチャージされる。２つのビットラインに少しの差がある間は、センスアンプによってプリチャージ状態が抑制される。十分な差ができる前にセンスアンプが開始すると、いくつかの非共通なノイズ、装置のミスマッチ、ビットラインプリチャージミスマッチによって、不正確な結果を導くということになってしまう。十分な差が現れるように時間が経った後に CLK がアサートする（１になる）。この場合 pulldown 装置 N1 が ON になるとセンスアンプの 2 つの脚から電流が流れ出していく。

CLK がセンスアンプを可動させる前に Mux__Out＃は 200mV より低くて、Mux__Out は高い状態を仮定する。N1 が導通した時、Sense と Sense#のノードは $V_{SS}$ のほうへ下がり始める。Mux__Out 信号が Mux__Out＃よりわずかに高い信号だから、センスアンプの左脚は右脚よりわずかに低い抵抗を持つだろう。この差はノード Sense よりノード Sense#をわずかに速く低くする。Sense#ノードが低くなると、P2 が ON になり N5 が OFF になり始める。これらの装置（P2、N5）は Sense の電圧が下がるのを防ぐため、Sense と Sense#の差が増加する。結局 P2 は Sense が $V_{DD}$ に近づくにつれて“ ON ”に強くなっていく。今の値をラッチするためにセンスアンプが生じている。ビットラインの差をひろげる操作は、CLK が変わるか、Sense#が０になるまで続けられる。

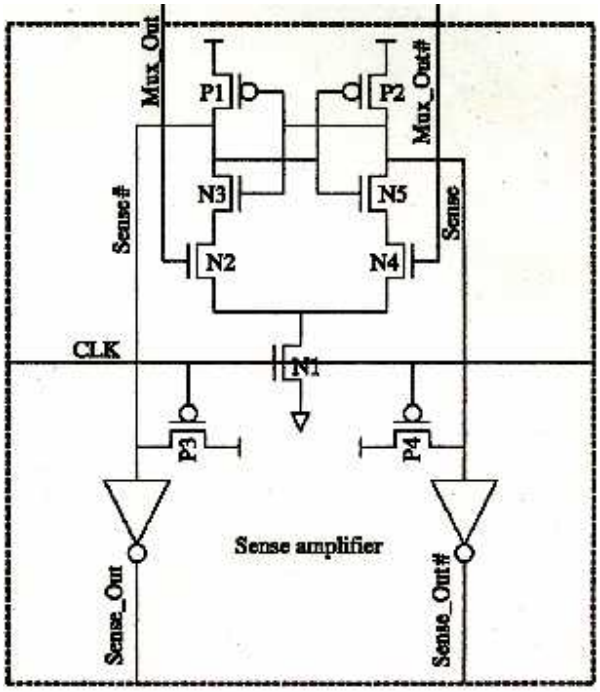


図２．５　センスアンプ

## おわりに

本研究では、FPU の加減算器の修正と LSI メモリの設計を研究した。

まず１章で、FPU の加減算器の修正を記した。まず、浮動小数点加算の概要を説明し、去年の加減算器の HDL 記述を記した。次に、シミュレーションによって加減算器のバグだしを行い、去年の加減算器の検証を行い、その検証を元に新加減算器のアルゴリズムを作成し、HDL で記述した。最後にシミュレーションを行い、新加減算器が正常に動作するか確認した。

２章では、メモリ LSI の回路設計を示した。まずメモリ LSI の分類を記し、次にメモリ LSI の基本構造と動作を記した。そして、DRAM、SRAM、プリチャージ方式、センスアンプについて説明した。

今後の課題としては、FPU のマルチポートレジスタの回路設計を行うこと、FPU にパイプライン処理を施すことによる性能向上をねらうことが可能だと思われる。

## 参考文献


【1】林成憲　町田慎弥　李天慧　國信茂郎：
ＨＤＬ（ハードウェア記述言語）を用いた浮動小数点プロセッサの設計

【2】パターソン&ヘネシー　日系 BP 社：
コンピュータの構成と設計　ハードウェアとソフトウェアのインタフェース
第 2 版　上・下

【3】株式会社エッチ・ディー・ラボ：
HDL Endeabor Verilog-HDL

【4】榎本忠儀　培風館：
CMOS 集積回路　－入門から実用まで－

【5】Anantha Chandrakasan・William J. Bowhill・Frank Fox　IEEE PRESS：
DESIGN OF HIGH-PERFORMANCE MICROPROCESSOR CIRCUITS