計算機アーキテクチャ　96－10
（1992. 10. 22）

ALUの例外処理の高速化手法

鈴木　一正　　山品　正勝　　山田　八郎
NEC　マイクロエレクトロニクス研究所　システムULSI研究部
神奈川県相模原市下九沢 1120

榎本　忠儀
中央大学　理工学部　情報工学科
東京都文京区春日 1-13-27

あらまし　マイクロプロセッサに用いられる演算器は，演算以外にも状態フラグの出力や演算結果の補正等の例外処理を行う．そのため，演算器を高速化するためには例外処理の回路を含めて高速化する必要がある．例外処理を高速化するためには，例外処理回路を演算器回路と並列にする方法が効果的である．そこで，例外処理回路の構成例として，桁上げ選択法を応用した零フラグ検出回路と，比較器を用いたオーバフロー補正回路を提案した．この回路を用いたオーバフロー補正機能と零フラグ検出機能を持つ 16-bit 加算器について，並列化の効果を見積ったところ，0.5 μm CMOS で処理時間が 4.20ns から 3.05ns に改善され，演算時間が 25% 短縮された．

和文キーワード　演算器　高速化　例外処理回路　零フラグ検出器　オーバフロー検出器

A methodology for a high-speed ALU with exception circuitry

Kazumasa Suzuki, Masakazu Yamashina and Hachiro Yamada
System ULSI Research Laboratory, Microelectronics Research Laboratories, NEC Corporation
1120, Shimokuzawa, Sagamihara, Kanagawa 229, Japan

Tadayoshi Enomoto
Department of Information and System Engineering, Faculty of Science and Engineering, Chuo University
1-13-27 Kasuga, Bunkyoku, Tokyo 112, Japan

Abstract

An arithmetic unit in a microprocessor not only has arithmetic operations but also has exception handling, for example flag detection and the result value correction. A parallel architecture for an arithmetic circuitry and an exception handling circuitry reduces the time required for exception handling, thus making the arithmetic unit faster. Examples of exception circuitry, a zero-flag detector and an overflow corrector are shown. These two circuitries reduce the operation time for a 16-bit adder with the zero-flag detection and overflow correction functions from 4.20ns to 3.05ns on a 0.5 micron CMOS technology.

英文 key words　arithmetic unit, high-speed methodology, exception handling, zero flag detector, overflow flag detector

1．はじめに

年々，マイクロプロセッサや信号処理プロセッサの処理の高速化が進んでいる．プロセッサはデータを取り込み，そのデータを演算し出力する処理を繰り返す．この処理を高速化するためには，高速な演算器が必要である．

演算処理の高速化については多くの方法が知られている[1]．例えば，加算処理の高速化には，桁上げ先見加算法や桁上げ選択加算法を用い，乗算処理の高速化には，Wallace のツリーや４入力２出力加算器ツリー[2]を構成し部分和を求めたり，冗長２進表現を用いて部分和を求めるときの桁上げ伝播をなくす[3]などの手法を用いる．しかし，演算器が行うのは演算処理だけではない．１つは，演算結果の性質を示す状態フラグ出力である．条件分岐などのプログラム制御命令がこの状態フラグの値を参照する．もう１つは，演算結果の補正である．演算結果を目的とする値の範囲内に収めるため，出力値に補正を加えることがある．演算器の処理を高速化するには，演算だけでなく，このような例外処理も高速化しなければならない．

本報告は，演算と例外処理の並列化によって例外処理の高速化を行うための手法について説明する．特に，並列化する処理回路の例として，オーバフローの補正回路，および，零フラグ検出回路の具体的な回路構成を提案する．さらに，この２つの回路を用いた演算器の処理時間を回路シミュレータで見積った結果を報告する．

2．例外処理の高速化

１章で例外処理には２つの種類があることを述べた．この章では，それらの２種類の処理を高速にする手法をそれぞれ説明した後，それらを実現した具体例として，オーバフロー補正回路，零フラグ検出回路の原理と回路構成を説明する．

2.1.例外処理の分類

例外処理の２つの型として状態フラグの検出と，演算値の補正がある．本報告では前者をフラグ型の例外処理，後者を補正型の例外処理と呼ぶことにする．

最初にフラグ型の例外処理であるが，これには演算結果が零であることを示す零フラグや，演算結果の正負の符号を示す符号フラグ，加減算で桁あふれを起こしたことを示すオーバフローフラグやキャリーフラグ，演算結果中の１の桁の数が偶数または奇数であることを示すパリティフラグ等の検出がある．オーバフローフラグやキャリーフラグのように加減算と同時に得られる状態フラグも一部にはあるが，他の状態フラグは演算結果から簡単な処理をすることによって得られるものである．したがって，回路構成を図１(a) のようにして，演算器の出力を例外検出器に入力して状態フラグを得る

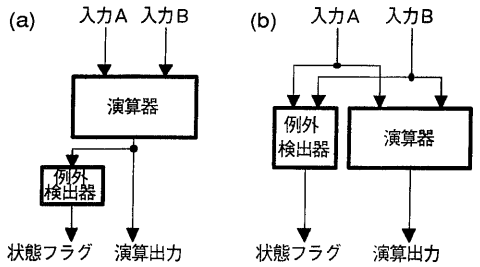

図１　補正型例外処理の回路構成 (a)従来 (b)並列方式

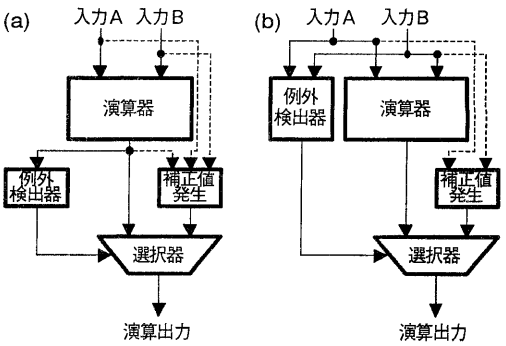

図２　補正型例外処理の回路構成 (a)従来 (b)並列方式

ことになる．例外検出器としては零フラグを得るには反転論理和 (NOR) 回路を，パリティフラグを得るには排他的論理和 (EXOR) 回路を用いればよい．しかし，この NOR 回路やEXOR 回路の入力信号数は演算器の出力信号のデータ幅に等しくなるため，32 ビットの演算器では 32 入力，64 ビットの演算器では 64 入力の NOR 回路や EXOR 回路を構成することになる．演算がさまざまな高速化手法によって処理時間を短縮できるようになり，この例外検出器の処理時間は無視できなくなった．そこで，図１(b) のように例外検出器を演算器と並列にし，入力した２値から直接演算結果の状態フラグを検出すれば，状態フラグを早いタイミングで得ることができ，演算器全体の処理時間を短縮できる．

次に補正型の例外処理であるが，これには演算結果がオーバフローを起こした場合に誤差を最小にするために，演算結果を最大値や最小値に補正するオーバフロー補正や，演算結果の正負によって，A-B と B-A の正となる方を出力をする絶対値演算回路等がある．この機能を実現させるには図２(a) のように演算結果と補正値を選択器に入力し，演算結果から例外を検出する例外検出器を通して，選択器を駆動するようにすればよい．補正値発生には必要ならば入力値や演算結果

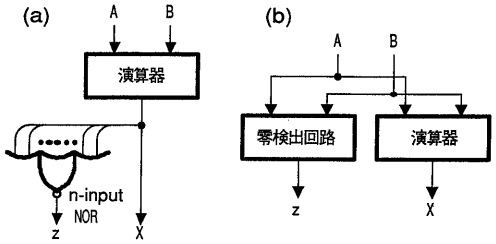

図3　零フラグ検出回路 (a)従来 (b)並列方式

等を入力し発生させる．オーバフロー補正では例外検出器にオーバフロー検出器を，補正値発生器には入力値の符号から最大値か最小値を発生させる回路を用いる．絶対値演算器では例外検出器に大小比較器を，補正値発生には減算器を用いる．しかし，この回路も，フラグ型と同様に例外検出器を演算器と並列に処理するようにできる．このような変更をしたのが図2(b) である．並列度を増すために，補正値発生も演算出力から求めるのでなく，入力値から直接求めるようにする．この変更によってフラグ型の時と同様な効果が得られ，処理時間を短縮できる．一方，選択器は出力のビット数分の1ビット選択器が並列になっていて制御入力のファンアウトが大きく，例外検出器の出力は大きな駆動力が要求される．図2(a) の回路では演算と同時に求められるオーバフローなどの出力の駆動力を上げるためにトランジスタサイズを大きくし最適化をするならば，演算器出力の負荷を増加させ，出力を遅くしてしまうことになり得る．そこで，図2(b) のように例外検出器を別回路にすれば，演算器と例外検出器を独立にトランジスタサイズの最適化ができ処理時間が削減される効果も期待できる．

2.2.零フラグ検出回路

フラグ型例外処理の例として演算結果が零となることを検出する零フラグ検出回路をとりあげる．前節で述べたように演算結果の全ビットの NOR をとれば零フラグを検出できる．これを示したのが図3(a) であり，処理時間を短縮させるには図3(b) のように零フラグ検出器を演算器と並列にすればよい．

演算には論理演算や加減算があるが，この中で処理に時間がかかるのは加減算である．加減算は例えば $(111...111)_2 + (000...001)_2$ の場合のように桁上げ信号を次々と上位に伝播させる処理が必要で，ビット毎の論理演算に比べどうしても回路のゲートの段数が多くなる．しかし，減算結果が零になるのは入力する2数が一致する場合であるから，各ビットの排他的論理和をとった結果が零になるかどうかを調べることで代用できる．そこで，加算結果の零フラグを検出する回路

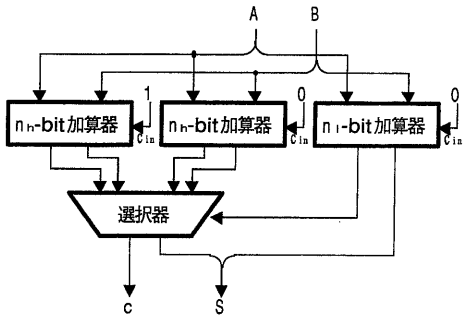

図4　桁上げ選択加算器

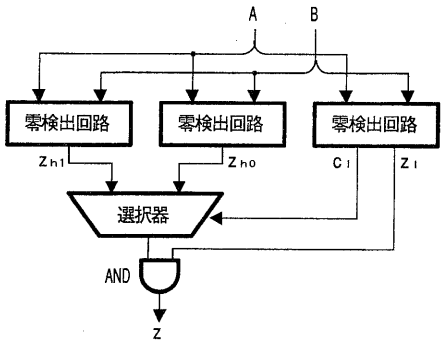

図5　零フラグ検出回路の原理図

を作成し，並列にして処理時間を短縮すればよい．

さて，加算は桁上げが生じるために演算に時間がかかることを述べたが，その演算を高速に行うための手法が何種類も考えられている．その1つに桁上げ選択法がある．これは長い語長の加算を短い語長に分割し，それぞれ独立に加算を行い，最後に結合する方法である．桁上げ選択法の加算器の構成例を図4に示す．入力する数をそれぞれ上位と下位に分割し，上位同士，下位同士を加算する．下位の加算結果は単純な加算の結果となるが，上位の加算結果は下位の加算結果の桁あふれの有無によって単純な加算となるか，それに1加えた値になるか不確定である．そこで，上位については2つの加算器を用いて2数の和と，2数の和に1加えた値を並列に求め，下位の桁上げ信号を制御信号とした選択器によって一方の値を選択出力する．もとの加算よりもビット幅が短い加算のため処理時間が短くできること，3つの加算をすべて並列に行えること，上位の加算結果の選択は各ビット毎独立に行えるので時間が加算ほどはかからないことによって，この加算器全体の処理時間はもとの長い語長の加算器よりも短縮できる．

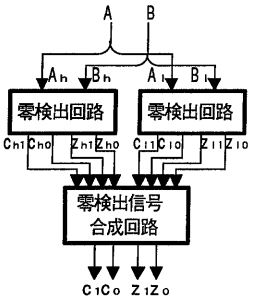

図6　零フラグ検出回路

これと同じことを零フラグ検出に用いる．その概念図を図5に示す．長い語長の加算結果の零フラグを求めるために，上位と下位に分割して短くなった2つの値についてそれぞれ独立かつ並列に零フラグを求める．この時，下位は加算の結果の零フラグ z_l の他に，上位への桁上げ信号 c_l を求め，上位は単純な加算の時の零フラグ z_{h0} と加算結果に1加えたときの零フラグ z_{h1} を求める．全体の零フラグは上位，下位共に加算が零となる場合であるから，下位から上位への桁上げがない場合，つまり，c_l が0の場合は z_{h0} と z_l の論理積を全体の零フラグとし，c_l が1の場合は z_{h1} と z_l の論理積を零フラグとすればよい．そこで，c_l を使って z_{h0} と z_{h1} の一方を選択器で選択し，その値と z_l を論理積すれば全体の零フラグとなる．この処理を式で表すと，

$$z = z_l \,\&\, ((\overline{c_l} \,\&\, z_{h0}) \,|\, (c_l \,\&\, z_{h1})) \qquad (1)$$

となる．ここで，& は論理積，| は論理和の演算を表す演算子である．

さて，これまでは上位と下位の2つのブロックに分割して，それぞれの零フラグを求めたが，分割後のブロックで零フラグを求めるのにも同じ方法を適用できる．下位となるブロックでは桁上げ信号を求めなければならないので，この桁上げ信号も零フラグと同様にそのブロック内の下位から上位への桁上げの有無に合わせた2通りを求めておき，桁上げ信号で選択できるようにする．上位となるブロックでは，結合後に下位からの桁上げの有無に対応する2通りの零フラグが必要であるので，そのブロック内の下位では下位ブロックからの桁上げに対応した2通りの桁上げ信号を発生させ上位の零フラグを選択し，下位の2通りの零フラグとそれぞれ論理積をとり2通りの零フラグを求める．このように，それぞれのブロックにおいて下位からの桁上げの有無に合わせて零フラグ2通り，桁上げ信号2通りの計4本の信号を求めておけば，次々にブロックを分割して零フラグを求められる．

以上に説明した零フラグ検出回路を図6に示す．これはブロックを分割し，それぞれの零フラグを並列に求める零検出回路2つと，それらの信号を結合し，全体の零フラグを求める零検出信号合成回路からなっている．ここで，下位ブロックの零フラグ信号を z_{l0}, z_{l1}，桁上げ信号を c_{l0}, c_{l1}，上位ブロックの零フラグを z_{h0}, z_{h1}，桁上げ信号を c_{h0}, c_{h1}，全体の零フラグを z_0, z_1，桁上げ信号を c_0, c_1 とする．但し，最後の添え字が0のものはその下位からの桁上げがない場合の値，1のものは桁上げ信号がある場合の値を表している．零検出信号合成回路の入出力の信号の関係を式 (1) をもとにしてまとめると，

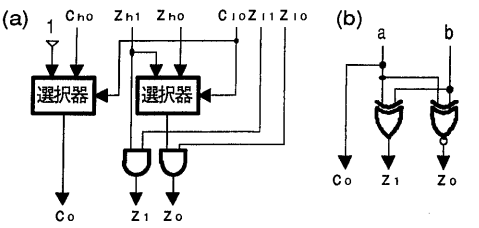

図7　零フラグ検出回路の構成要素
(a)零検出信号合成回路 (b) 1ビット零検出回路

$$z_0 = z_{l0} \,\&\, ((\overline{c_{l0}} \,\&\, z_{h0}) \,|\, (c_{l0} \,\&\, z_{h1})) \qquad (2)$$

$$z_1 = z_{l1} \,\&\, ((\overline{c_{l1}} \,\&\, z_{h0}) \,|\, (c_{l1} \,\&\, z_{h1})) \qquad (3)$$

$$c_0 = (\overline{c_{l0}} \,\&\, c_{h0}) \,|\, (c_{l0} \,\&\, c_{h1}) \qquad (4)$$

$$c_1 = (\overline{c_{l1}} \,\&\, c_{h0}) \,|\, (c_{l1} \,\&\, c_{h1}) \qquad (5)$$

となる．

さらに，分割を進めると，最終的に1ビットの零検出器に到達する．1ビットの場合は，零フラグと桁上げ信号を簡単な論理ゲートで求められる．下位からの桁上げがない場合は，2数がともに0の時と，ともに1の時に和が0となるため，2数の排他的反転論理和で零フラグが得られる．桁上げ信号は，2数がともに0の時には生じず，ともに1の時は生じる．一方が1で他方が0の時は，下位からの桁上げがそのまま桁上げ信号になるのだが，この時の零フラグは0であるため，上位の零フラグの値に関わらず零検出信号合成回路で合成した零フラグは必ず0になり，正確な桁上げ信号を発生させる必要がない．そこで，2数の一方の値をそのまま桁上げ信号としてしまえば十分である．下位からの桁上げがある場合は，2数の一方が0，他方が1の時に和が0となるため，2数の排他的論理和で零フラグが得られる．零フラグが1となる入力の組合せでは桁上げ信号が常に1となる．その他の入力の組合せでは零フラグが0であるため，先ほどと同様な理由によって桁上げ信号を正確にする必要がなく，桁上げ信号は常に1にしておけば十分である．これらをまとめると，

$$z_0 = \overline{a \wedge b} \qquad (6)$$

$$z_1 = a \wedge b \qquad (7)$$

$$c_0 = a \ (\text{または} b) \qquad (8)$$

$$c_1 = 1 \qquad (9)$$

となる．ここで，a, b は1ビットの2数，∧は排他的論理和の演算子である．

ところで，式(9) から，1ビット零検出器の c_h は常に1であるから，式(3) と式(5) は更に簡単な形にできる．式(3) と式(5) の c_{11} と c_{h1} に1を代入すると，

$$z_1 = z_{11} \& z_{h1} \qquad (3')$$

$$c_1 = 1 \qquad (5')$$

となる．このように,c_1 は常に1となるため，次に結合するときにも式(3') と式(5') が使える．このことを考慮して，零検出信号合成回路の具体的な回路を図7 (a) に，1ビットの零フラグ検出器回路を図7 (b) に示す．1ビットの零検出回路を零検出信号合成回路で結合し，更に，結合を続ければ任意ビット幅の零フラグ検出回路が構成できる．

2.3.オーバフロー補正回路

補正型の例外処理の例としてオーバフロー補正回路をとりあげる．従来，加減算のオーバーフローを検出するには，演算結果の数値全体からの桁上げ信号と，符号部を除いた数値部からの桁上げ信号の一致を排他的論理和を用いて検出していた．この方法では，加減算の結果とほぼ同時にオーバフローが検出できる．

プロセッサの応用によってはオーバフローを起こした場合，最大値や最小値に値を補正し演算誤差を最小にする演算器を持つことがある．この機能を実現するには図8 (a) のような回路構成をとればよい．オーバフロー補正を行う場合，負荷が大きい選択器の制御入力をこのオーバフロー出力が駆動しなければならず，強力なバッファを用いるので処理時間が増大してしまう．そこで，図8 (b) のようにオーバフロー検出器を演算器と並列にし，オーバフロー検出器内で，トランジスタサイズの最適化をし，駆動力を大きくする．また，オーバフロー検出器が演算よりも短時間の処理で行えれば全体の処理時間を短縮できることが期待できる．

そこで，入力数から直接加減算のオーバフローを検出する回路を，比較器を用いて構成する．最初に，nビットの2の補数表示数 A, B の加算のオーバフローを考える．nビットの2の補数表示数が表せる範囲は $2^{n-1}-1$ から -2^{n-1} であるから，A+B がオーバフローする条件は，

$$A+B > 2^{n-1}-1 \quad (A, B \geqq 0) \qquad (10)$$

$$A+B < -2^{n-1} \quad (A, B < 0) \qquad (11)$$

である．図9 (a) は4ビットの場合について，A+B がオーバフローするAとBの組の座標に×を記入したものである．4ビットでは式(10) の右辺は7，式(11) の右辺は -8 であるから，和が8以上の部分および -9 以下の部分がオーバフローとなる領域である．この図からもわかるように，演算後のオーバフロー領域は，ともに符号ビットが0となる A≧0 かつ B≧0 の場合と，ともに符号ビットが1となる A<0 かつ B<0 の場合に限られる．

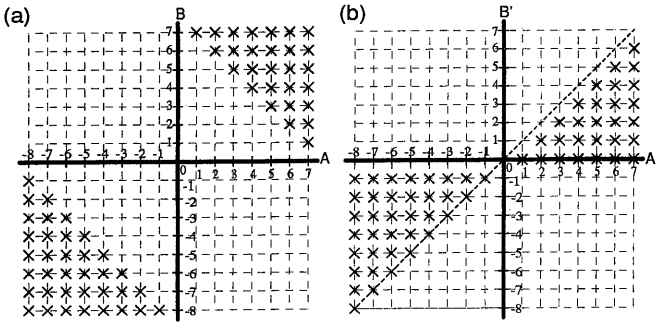

図9　4ビット加算 A+B でオーバフローを起こす範囲
(a)変換前　(b) Bの符号以外ビット反転後

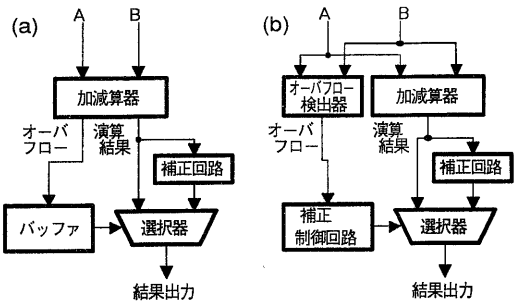

図8　オーバフロー補正回路の構成 (a)従来 (b)並列方式

ここで，オーバフロー領域を比較器で検出できる形に変換することを考える．Bの符号ビットはそのままにして，符号以外のビットを反転する操作をする．この操作後のBをB'とする．nビットの2の補数 $B=\{b_{n-1} b_{n-2} \cdots b_1 b_0\}$ は，整数

$$B = -b_{n-1} \cdot 2^{n-1} + \sum_{i=2}^{n-2} b_i \cdot 2^i \qquad (12)$$

を表すから，BとB'の関係は，$b_{n-1}=0$ つまりB≧0 のときは，

$$\begin{aligned} B+B' &= \sum_{i=2}^{n-2} b_i \cdot 2^i + \sum_{i=2}^{n-2} \overline{b_i} \cdot 2^i \\ &= \sum_{i=2}^{n-2} (b_i + \overline{b_i}) \cdot 2^i \\ &= \sum_{i=2}^{n-2} 1 \cdot 2^i \\ &= 2^{n-1} - 1 \quad (B \geqq 0) \end{aligned} \qquad (13)$$

となり，また，$b_{n-1}=1$ つまり $B<0$ のときは，

$$B+B' = -1\cdot 2^{n-1} + \sum_{i=2}^{n-2} b_i \cdot 2^i - 1\cdot 2^{n-1} + \sum_{i=2}^{n-2} \overline{b_i}\cdot 2^i$$
$$= -2\cdot 2^{n-1} + \sum_{i=2}^{n-2} (b_i + \overline{b_i})\cdot 2^i$$
$$= -2\cdot 2^{n-1} + \sum_{i=2}^{n-2} 1\cdot 2^i$$
$$= -2\cdot 2^{n-1} + 2^{n-1} - 1$$
$$= -2^{n-1} - 1 \quad (B<0) \qquad (14)$$

となる．Bが正の場合は，式(13) を式(10) に代入して，

$$A+(2^{n-1} - 1 - B') > 2^{n-1} - 1$$
$$\therefore \quad A > B' \qquad (15)$$

となり，またBが負の場合は式(14) を式(11) に代入して

$$A+(-2^{n-1} - 1 - B') < -2^{n-1}$$
$$\therefore \quad A < B'+1 \quad \text{または} \quad A \leqq B' \qquad (16)$$

となり，この範囲でオーバフローを生じることがわかる．図9 (b) に4ビットの場合の変換後のオーバフロー領域を示す．以上まとめると，加算 A+B のオーバフローの領域は，

A≧0, B≧0 の範囲で A>B',

A<0, B<0 の範囲で A≦B'

である．したがって，オーバフローは2数 A, B の符号と A, B' の大小関係で判定できる．

次に，減算のオーバフローを考える．減算は減数を2の補数変換することにより，正負の符号を反転し加算して行う．2の補数表現数Bを2の補数変換するには $\overline{B}+1$ を求めればよいから，A-B を行うには $A+\overline{B}+1$ を計算することになる．そこで，A+B+1 の場合のオーバフローを A+B のときと同様に考える．A+B+1 がオーバフローする条件は，

$$A+B+1 > 2^{n-1} - 1 \quad (A, B \geqq 0) \qquad (17)$$
$$A+B+1 < -2^{n-1} \quad (A, B < 0) \qquad (18)$$

と表される．4ビットの場合のオーバフローする範囲を図9 (a) と同様に記した図を図10(a) に示す．

A+B のときと同様に，Bの符号をそのままにし，符号以外のビットを反転する操作を行う．Bと変換後の B'の関係は式(13) と式(14) に示すとおりである．そこで，B≧0 の場合，変換後のオーバフロー領域は式(13) を式(17) に代入して，

$$A+(2^{n-1} - 1 - B') + 1 > 2^{n-1} - 1$$
$$\therefore \quad A > B' - 1 \quad \text{または} \quad A \geqq B' \qquad (19)$$

となり，また，B<0 の場合は式(14) を式(17) に代入して，

$$A+(-2^{n-1} - 1 - B') < -2^{n-1}$$
$$\therefore \quad A < B' \qquad (20)$$

となる．図10(a) に対してこの変換を行った場合のオーバフロー領域を図10(b) に示す．

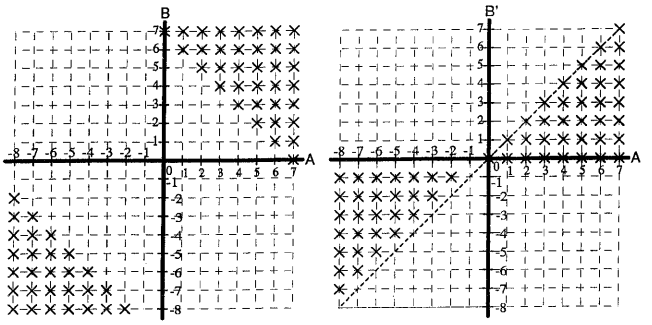

図10　4ビット加算 A+B+1 でオーバフローを起こす範囲
(a)変換前　(b) Bの符号以外ビット反転後

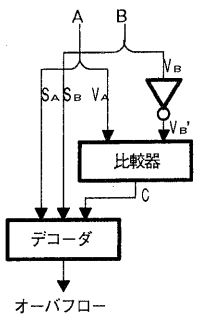

図11　オーバフロー検出器
A+B 専用

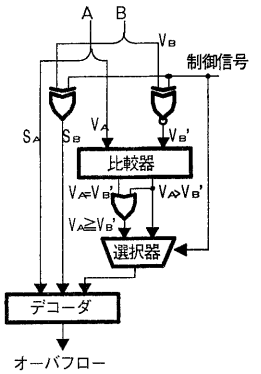

図12　オーバフロー検出器
A+B, A-B 兼用

以上まとめると，加算 A+B+1 のオーバフローの領域は，

A≧0, B≧0 の範囲で A≧B',

A<0, B<0 の範囲で A<B'

で与えられる．したがって，A+B+1 のオーバフローも A+B と同様に2数 A, B の符号とAと B'の大小関係で判定できる．

以上述べてきたように，加算のオーバフロー検出は符号ビットの値と，符号以外の部分同士の比較結果を用いて検出することができる．そこで，ビット毎の反転を求める反転器(NOT) と，大小比較を行う比較器，及び2数の符号と比較器の結果からオーバフローの判別を行うデコーダでオーバフロー検出器を構成できる．図11 にオーバフロー検出器の回路構成を示す．入力数AとBは符号ビット S_A, S_B と符号以外ビット V_A, V_B に分けられ，V_A はそのまま比較器に，V_B 側では NOT でビット反転した値 V_B' を比較器に入力する．比較器は V_A と V_B' の大小比較を行い，$V_A>V_B'$ ならC=1 を $V_A\leqq V_B'$ なら C=0 を出力する．2の補数表示数では $S_A=S_B$ の範囲で V_A と V_B' の大小関係がAとB' の大小関係と等しい．よっ

て，オーバフローを起こす A, B'ともに0以上，もしくは A, B'ともに負の範囲では，V_A と V_B の大小関係を求めればAとB'の大小関係を求めるのに十分である．

デコーダには2数の符号 S_A, S_B と比較器の出力Cが入力される．これらの信号から，オーバフローを起こす条件となる組合せつまり，

S_A=0, S_B=0, C=1,

S_A=1, S_B=1, C=0

の組合せの場合に1を出力する回路をデコーダとして用いれば A+B のオーバフロー検出器となる．

出力Cを V_A<V_B' で0，V_A≧V_B' で1となる比較器を用いれば，A+B のオーバフローとなる条件と同じ S_A, S_B, C の組合せで A+B+1 もオーバフローとなるため，ほとんど同じ回路構成で A+B+1 のオーバフロー検出器を構成することができる．A+B と A+B+1 のオーバフローが同一のデコーダ回路で行えるので，比較器の部分を工夫することによって加減算両方に対応するオーバフロー検出器も構成できる．このようなオーバフロー検出器を図12 に示す．比較器は A>B, A=B を出力するようにし，1つはA>B をそのまま，もう1つはA>B と A=B の論理和をとり A≧B とすることができるので，選択器でその一方を選択しデコードすれば A+B, A+B+1 のオーバフロー検出器となる．比較器の出力と同時に，入力側の反転器も同じ制御信号でBの側の反転，非反転を制御すれば A+B と A-B のオーバフロー検出器となる． 図12 では EXOR と EXNOR でこの制御を行っている．このオーバフロー検出器を図8の補正回路中のオーバフロー検出器のブロックに使用すればオーバフロー補正回路が構築できる．

3．本報告の例外検出法の効果

2章では例外処理の例題として零フラグ検出とオーバフロー補正処理の高速化について説明した．この2つの例外処理を持つ算術論理演算器を想定し遅延時間を比較した．その結果について述べる．

目標とする回路と機能は，16-bit の2の補数の加算を行う演算器で，オーバフロー補正機能と零フラグ出力を持つものとする．オーバフロー補正は，正にオーバフローする場合は正の最大値 $(0111111111111111)_2$ を出力し，負にオーバフローする場合は負の最小値 $(1000000000000000)_2$ を出力する．零フラグ出力は演算結果が0となった場合に1をそれ以外のときは0を出力するものである．

従来の方法でこの機能の演算器を作成すると図13(a) のような構成となる．演算器が演算結果と同時にオーバフローを検出し，その信号を使って選択器を駆動し演算結果か補正値かを選択する．その結果を受けて，NOR で作られた零フラグ検出器が零フラグを出力する．一方，本報告で述べた回路を使用する場合の回路構成は図13(b) のようになる．演算器と零フラグ検出器，オーバフロー検出器が並列になり，オーバフロー検出器の出力で選択器を駆動する．選択器は演算結果と補正値の一方を出力する．

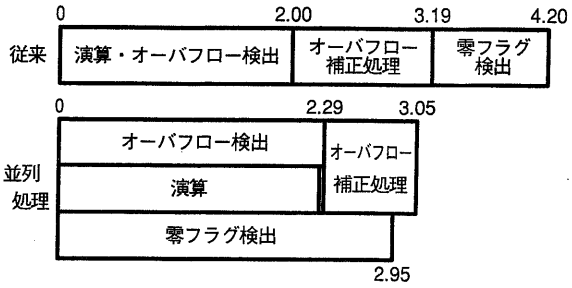

図14　演算処理時間

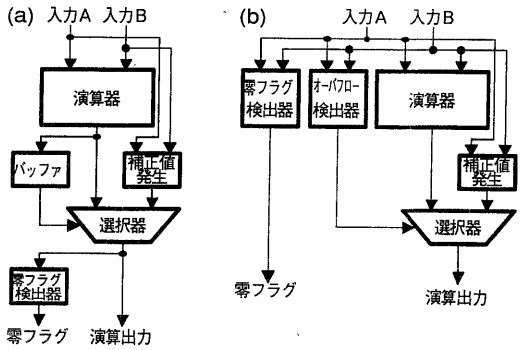

図13　シミュレーションに使用した演算器の回路構成
(a)従来方式 (b)並列方式

演算器を CLA を改良した回路を用いて構成し，オーバフロー検出器の比較器を2ビット毎まとめてツリー構成にした比較器を用いるとして，SPICE シミュレーションによって演算処理時間を求めた結果を図14 に示す．使用するプロセスパラメータは 0.5 μm BiCMOS プロセスのものである．ただし，回路はすべて CMOS を使用している．従来の方法ではオーバフロー補正までに 3.19ns，零フラグ検出までには 4.20ns かかっている．一方，並列処理することによって，零フラグ検出までが 2.95ns，オーバフロー補正出力までが 3.05ns に短縮される．このように，並列化したことによって約 25% 処理時間が短縮された．また，並列化することによってハードウェア量が増加するが，従来の回路は約 2000 トランジスタ，並列処理した回路は約 3100 トランジスタとなり，5割程度の増加となる．

4．まとめ

プロセッサに使われる演算器は演算以外にもフラグの出力や，値の補正などの例外処理を行う．演算器を高速化するためには例外処理の回路を含めて高速化しなければならない．そのためには例外処理回路を演算器回路と並列にする方法が効果的である．例外処理回路の構成例として，桁上げ選択法を応用した零フラグ検出回路と，比較器を用いたオーバフロー補正回路を紹介した．この回路を用いたオーバフロー補正機能と零フラグ検出機能を持つ 16-bit 加算器について，並列化の効果を見積ったところ，処理時間は 25% 減らすことができた．

謝辞

このような研究の機会を与えてくれた NEC マイクロエレクトロニクス研究所の渡辺所長，岡田所長代理，システムULSI研究部の高田部長をはじめとする関係者に感謝致します．

参考文献

[1] 堀越 彌，"コンピュータの高速演算方式"，近代科学社，1980.

[2] 永末他，"3 層金属配線を使用した32ビット並列乗算器"，信学技報 ICD89-128, pp27-33, 1989.

[3] M.Yamashina et al., "A 200-MHz 16-bit Super High-Speed Signal Processor (SSSP) LSI", IEEE Jounal of solid-stage circuits, Vol.24, No6, Dec. 1989.