FPGA を用いた高効率・高速数値演算

瀬尾雄三†

概要：多数の演算器を用いるパイプライン演算装置は，論理回路の規模が大きくなることが欠点であるが，それぞれの演算が異なる演算器でおこなわれることから，各演算器を必要最小限の規模に構成すればこの問題は回避される．シグナル・プロセス・ロジック社は，有用なビットのみを演算して最小の数値演算論理を形成する数値演算論理開発環境を提供している．本報では，この基本思想と技術内容について解説するとともに，従来の CPU を用いる計算機との違いについて考察し，今後の方向を探りたい．

キーワード：FPGA，パイプライン，有効ビット，数値演算

High-efficiency and High-speed Numerical Computation using FPGA

YUZO SEO†

Abstract: Pipeline computer has a disadvantage of large circuit scale since large number of computing units are used. However, since the respective operations are performed in different operation unit, this problem is avoided by configuring to the minimum necessary size of each arithmetic unit. Signal Process Logic Inc. provides a numerical arithmetic logic development environment that forms the smallest numerical arithmetic logic by treating only the useful bits. In this article, we describe the basic idea and techniques, discuss the differences to traditional computer using CPU, and search the future directions.

Keywords: FPGA, pipe line, effective bits, numerical computation

1. はじめに

パイプライン演算装置は，多数の演算器が並列動作することから，高速演算が可能という特徴をもち，画像処理やストレージ装置の信号処理などに多用されている．このためのデバイスとして，従来は ASIC が用いられていたが，近年では論理書き換えが可能な FPGA も広く利用されるようになり，小ロット分野にも応用範囲を広げている．

FPGA を用いた数値演算に際しては，デバイスの持つ論理エレメントが限られることから，演算論理の縮小が強く求められる．科学技術計算においては，値の精度が重視され有効桁のみを表示することが一般的であるが，機械計算においても，有効桁のみを処理すれば演算に必要な論理資源は大幅に削減することができる．

これを実現するための個々の技術に関しては，これまでも発表をおこなってきた．有効桁に着目した演算論理縮小に関しては 2010 年に[1]，データフロー型の演算論理を記述する言語仕様に関しては 2012 年に[2]，有効桁のみを扱う際に問題となる誤差のキャンセルと対応方法に関しては 2014 年に[3]それぞれ発表している．また，この技術を応用した数値演算論理設計支援ツールを開発し評価版を公開している[4]．これらの技術要素の発表とツール評価版公開の経緯は表 1 のとおりである．

本報では，これらの要素技術とこれを用いた数値演算論理設計支援ツールを総合的に紹介するとともに，ストアードプログラム方式の数値演算装置へと展開するためのデバイス技術を提案し，現在の技術水準でも FPGA をベースとした高効率・高速数値演算装置が可能であることを示す．また，従来の CPU を用いる数値演算との相違点について考察し，今後の方向を探ることとしたい．

表 1 要素技術と評価版ツールの発表経緯

Table 1. History of the Elemental Technologies and Tools

名称	有効桁処理	コンパイラ	誤差相殺対応	技術発表	デモ・評価版リリース[4]
CodeSqueezer Basic	☐	☐	☐	EDS Fair 2010	2010/1/22
—	☑	☐	☐	DAS 2010[1]	
CodeSqueezer Floating	☑	☑	☐	EDS Fair 2011	2010/11/4
CodeSqueezer	☑	☑	☐	DAS 2012[2]	2011/10/31
CodeSqueezer 2.0	☑	☑	☑	情報処理学会全国大会2014[6] DAS 2014[3]	計画中

2. 数値演算論理の自動形成

2.1 ブロックダイアグラム

パイプライン型の演算器は，多数配置された演算器をデータが順次通過するデータフロー型の演算装置として構成される．このような装置は，基本演算機能をもつ多数のブロックを信号線（リンク）で接続したブロックダイアグラムとして表現される．

コンパイラは，ユーザが数式を用いて記述した演算仕様

† シグナル・プロセス・ロジック(株)
Signal Process Logic Inc.

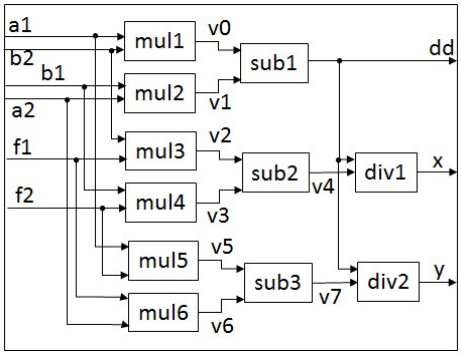


図 1　ブロックダイアグラム

Figure 1　Block Diagram

を，ブロックダイアグラムに変換する．ブロックダイアグラムはそれ自体が一つのブロックであり，上位のブロックダイアグラムを構成する要素として用いられる．四則演算などの基本的な演算機能をもつブロックは，標準機能ブロックとして，システムにあらかじめ用意されている．

ブロックダイアグラムの一例を図 1 に示す．このブロックダイアグラムは，ユーザが次式（クラメルの公式）を与えることでコンパイラにより自動的に形成される．

dd = d = a1 * b2 – b1 * a2

x = (f1 * b2 – b1 * f2) / d

y = (a1 * f2 – f1 * a2) / d

図中，mul は乗算，sub は減算，div は除算の標準機能ブロックであり，使用される毎に配置される．それぞれの矢印をもつ線はリンクと呼ばれる信号線である．これらは，ブロックダイアグラムの段階では数値型をもたず，数値情報が伝達される抽象的な信号線を意味する．

2.2 信号線の数値型と型属性

有効桁のみを処理するためには，浮動小数点表現が必要となる．今日の浮動小数点表現には IEEE754 の規定する形式が多く用いられているが，FPGA 上に形成される演算論理は 1 bit 単位での論理形成が可能であることから，より柔軟性の高い数値型を用いる．また，FPGA 上に実装される論理を最小とし高速化を可能とするため，数値型もこれに適したものとする[1].

数値は，仮数部と指数部を表す二つの整数（M および X）を用いて，値を $M \times 2^X$ で表す．負数の表現は二の補数による．異常の表示には，値が表現可能範囲を上に越えたことを示すオーバーフロー（O），下に越えたことを示すアンダーフロー（U），数値化不能（N）の三つの整数を別途用いる．これらは 0 または 1 の値をとる．

それぞれの整数の値は，FPGA 上に実装される信号線の値と定数の和で与える．整数の属性は，この定数値と，信号線の値の最大値・最小値で規定し，最大値と最小値に応じてこれを伝達するに必要な信号線を FPGA 上に形成する．最大値と最小値がともにゼロであれば，その整数は定数であり，FPGA 上には信号線を形成しない．

このような数値型は，演算論理の最小化を可能とするだけでなく，数値型（整数，固定小数点数，浮動小数点数，これらの変数および定数，符号の有無）を意識しない，抽象度の高いプログラミングを可能とする．

数値には，誤差を含むものと誤差を含まないものの二種類がある．これらを識別するため，誤差を含むか否かを数値属性として与える．コンパイラは，誤差をもつ変数に対しては指定された数の無効桁が含まれるように演算論理を形成し，次節で述べるノードグラフ形式に変換する．後段への無効桁の伝達は，フィルタリングなどの平均化処理をおこなう場合や誤差のキャンセルが発生する場合などに意味をもつ．後者に関しては 2.5 節で述べる．

2.3 ノードグラフ

ノードグラフは，Verilog HDL の代入文に対応するノードを，整数に対応する信号線（ワイヤ）で接続したものである．代入文の右辺には，単一の信号線，二つの信号線間の加減算・乗算・論理演算・結合，三つの信号線からなる条件式のいずれかがおかれる．

右辺に現れる信号線には修飾を施すことができる．修飾には，ビットの拡張および縮小，リダクション演算，符号反転，論理化および否定論理化がある．左辺に現れる信号線は，必要に応じて自動的にレジスタが割り当てられる．

2.4 最適演算論理の形成

標準機能ブロックには，それぞれの機能を実現するノードグラフを生成するコード化関数が準備されている．コード化関数は，入力ポートに接続されたリンクの属性に基づいて最適な演算論理を作成し，出力ポートに接続されたリンクの数値型属性を設定する．

有効桁のみを処理する論理を形成するための手法には，誤差分散を管理してコードを形成する手法と，有用な情報を含む桁を保存するコードを形成する手法の二通りがあるが，現在後者の手法を採用している．この手法は，演算を繰り返す過程で演算結果に含まれる誤差が増大する問題がある半面，情報の損失が少ない利点がある．誤差増加の問題に対しては，分解能を縮小する標準関数を用意し，設計者の判断で適宜これを挿入して対処することとした．

2.5 誤差キャンセルによる問題と対応

指数部が大きく異なる値の加減算では，指数部の小さな値に含まれる情報が失われる．この演算結果に対して後段で逆の演算がおこなわれ，指数部の大きな値に含まれる誤差がキャンセルされても，失われた情報は回復されない．後段で誤差のキャンセルがおこなわれる場合には，有効桁以下のビットも後段に伝達する必要がある[3].

たとえば，次の二次方程式において，

$$x^2 - 2bx + c = 0 \quad (1)$$

$$x = b \pm \sqrt{b^2 - c} \quad (2)$$

(1)式の解は(2)式で与えられるが，係数 b に含まれる誤差は

b が負の場合は加算時に，正の場合は減算時にキャンセルされる．このため，b の指数部が c の指数部に比べ非常に大きい場合には，平方根内の減算により c に含まれる情報が失われ，本来計算されるべき値を得ることができない．

倍精度浮動小数点数を用いる一般的な演算に際しては，有効桁をはるかに超えるビット幅を用いて演算をおこなうためこの問題は表面化しにくいが，有効桁のみを処理する演算論理では誤差のキャンセルが直ちに問題を引き起こす．

誤差キャンセルの問題は，式の変形によっても回避可能である．すなわち，(2)式を(3)式に変形すればよい．

$$x = c / (b \mp \sqrt{b^2 - c}) \qquad (3)$$

こうすることで，b の符号が(2)式とは逆の場合に誤差のキャンセルが発生する．したがって，b の符号に応じていずれの計算式を用いるかを選択すればよい．

しかしながら，種々の関数が使用される一般的な数式に対してこのような処理を自動的におこなうことは難しい．そこで，数式は固定したまま，誤差のキャンセルが生じた場合に有用となる無効桁を後段に伝達することとした．

誤差キャンセルの発生は，入力された値の大小関係に依存する．そこで，演算論理の形成に先立って，可能な入力値の全組合せ範囲にわたって演算過程をシミュレートすることにより誤差キャンセルを検出し，それぞれのブロックの出力に要求される無効桁数を設定する．

(2)式を演算するブロックダイアグラムを図 2 に示す．入力変数は二つの係数 b および c であり，それぞれが誤差 e_b および e_c をもつ．シミュレーションの段階では，リンクは値のほかに，要因別の誤差の大きさを要素とするベクトルをもつ．図 2 に示したように，sub2 の出力リンクの誤差ベクトルの第一要素には減算が含まれており，誤差のキャンセルが発生し得る．

信号線の値の誤差分散は，要因毎の誤差の二乗和で与えられる．ブロックの無効桁要求と誤差分散から出力の仮数部桁数が決まる．シミュレーションは入力される b と c の組合せに対してそれぞれの信号線の値と値に含まれる誤差分散および要因別の誤差の値を計算し，出力に要求される桁数を計算するに十分な桁数が入力されているか否かをチェックする．そして，これが不足する場合にはその信号を

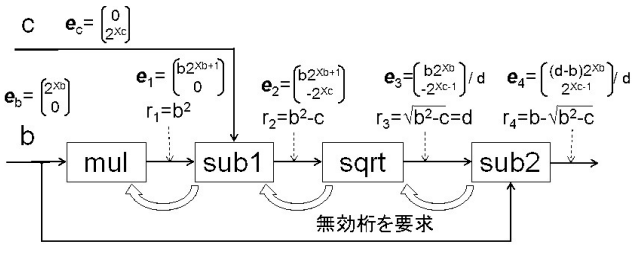


図 2　二次方程式の解を求めるブロックダイアグラム

Figure 2　Block Diagram to Solve Quadratic Equation

出力している前段ブロックの無効桁要求を増加し，そのブロックと，この出力に接続されている処理済みのブロックの処理済みフラグを再帰的に未処理状態に戻す．この演算を，未処理ブロックに対して繰り返しおこなうことで，すべてのブロックに要求される無効桁数が決定される．

3. 数値演算記述言語 mhdl

3.1 コーディング規約

mhdl は数式を記述する簡素な言語であり，文法に従って数式を記述するだけでデータフロー型の演算装置を構成することができる[2].

mhdl の文は，ブロック宣言文と代入文の二種類のみである．ブロックは C 言語における関数に相当する．

mhdl は，今日ソースコードを記述する際に広くおこなわれている字下げに意味を持たせることで，ブロックの範囲を規定するためのキーワードや記号を不要としている．文の行頭にあるブランクの数を「文のレベルの深さ」と呼び，ブロック宣言文は，この文のレベルよりも深いレベルの文が続く間におかれた代入文とブロック宣言文を含む．

行末は文の区切りとする．ただし，演算子，コンマまたは開きかっこのいずれかで終わる行は次行に継続する．コメントは，C 言語と同様，“//” 以降行末まで，もしくは “/*” と “*/” に挟まれた部分に記述する．コメントおよび行末の空白は処理に先立って除去される．

名前（変数名およびブロック名）は，C 言語と同様，英字に始まる任意字数の英数字とする．アンダースコア “_” は英字と同様に扱われるが，システムが生成する名前の先頭に使用しているため，名前の先頭に用いてはならない．

3.2 ブロック宣言文

ブロック宣言文は以下の構文からなる．

name.(out1, out2, ...) (in1, in2, ...)

“name” はブロック名，“out1, out2, ...” は出力ポート名，“in1, in2, ...” は入力ポート名である．出力ポート名が単一の場合は出力変数名を囲むかっこは不要である．出力ポート名とその前のピリオッド “.” を省略した場合はブロック名が出力ポート名となる．入力ポート名を囲むかっこは省略できない．出力される値は，出力ポート名を左辺とする代入文により設定される．

ブロック名は，そのブロック宣言文がおかれたブロックの範囲内で有効である．再帰呼び出しはできない．

最上位ブロックのブロック宣言文は省略することができ，その場合は，代入文の右辺にのみ現れる変数名が入力ポートとして，代入文の左辺にのみ現れる変数名が出力ポートとして定義されたものとみなされる．この規定により，数式のみの記述も mhdl のソースコードとなる．

3.3 代入文

代入文は，最も左側の演算子を代入演算子とするもので，任意の数の二項演算子で区切られた項を並べることができ

表 2. 二項演算子

Table 2. Binary Operators

優先度	演算子	演算内容	項数	演算順序
高 ⇕ 低	^	冪乗	任意	右→左
	*, /	乗算，除算		不定
	+, -	加算，減算		
	<, <=, == !=, >=, >	比較	2	–
	&	論理積	任意	不定
	!	排他論理和		
	|	論理和		
	?, :	条件式	3	–
	=	代入	任意	右→左

る．ただし，代入演算子の左には，代入演算子以外の演算子があってはならない．

二項演算子を表 2 に示す． C 言語と異なる点は，べき乗演算子“^”を追加したこと，排他論理和の演算子を“!”としたことである．また，mhdl の式にはビットごとの演算が存在しない．これにともない，論理演算子（“&”，“|”，“!”）は一文字とした．なお，現在公開中のコンパイラはべき乗演算子をポートしていない．

項の前には，単項演算子を付けることができる．単項演算子には，符号反転演算子“-”，符号非反転演算子“+”，論理化演算子“!!”および否定論理化演算子“!”がある．単項演算子は二項演算子に優先して作用する．演算子が連続する場合は空白で区切る．

データフロー型の演算装置では，同じ信号線に値を設定しなおすことができない．このため，C 言語とは異なり，インクリメント・デクリメント演算子は存在せず，演算と代入を単一の演算子でおこなう複合代入文も存在しない．

式をかっこ（“(”と“)”）でくくったものは項として扱われ，かっこ内が優先して演算される．

定数も項として扱われる．定数は数字の並びで記述され，先頭以外に一つの小数点を含むことができる．また，指数文字に続く数字の並びにより指数部を与えることができる．数値は十進数と解釈される．小数点もしくは指数文字“E”を含む定数は誤差あり定数を，指数文字“D”または“B”を含む定数は誤差なし定数を意味する．“B”に続く数値は底を 2 とする指数部を，その他は底を 10 とする指数部を示す．“D”に続く指数部は負であってはならない．

代入演算子の左辺には，変数名のリストをおくことができる．リストは複数の名前をコンマ“,”で区切って並べたものをかっこでくくったものである．変数名のリストへの代入機能は，複数の出力ポートをもつブロックから値を得る際に用いられる．

代入文の右辺におかれる変数は，それ以前に代入文左辺または入力ポート部に表れていなければならない．この規定により，データが一方向に流れることが保証される．

4. CodeSqueezer

4.1 CodeSqueezer の機能

CodeSqueezer は，組込みシステム用の数値演算モジュールを簡単な操作で作成するためのツールで，前節までに述べた手法に基づいて，mhdl 形式で記述された数式を Verilog HDL で記述された演算モジュールに変換する．また，必要な個所にレジスタを挿入してタイミング制約を解決する機能と，生成された Verilog HDL コードの動作確認機能をもつ．これらは対話形式のボタン操作で遂行される[4].

4.2 CodeSqueezer の操作例

ここでは，以下の演算式を与えて，浮動小数点加算器を作る例について紹介する．

floating_add(in1, in2)

floating_add = in1 + in2

ソフトウエアを起動すると図 3 の起動画面が現れる．演算式は，open ボタンによりファイルから読み込むか，file メニューの new を選択して新たに画面上で作成する．

演算式設定後 compile ボタンを押すと，図 4 に示す外部入力の型設定画面が現れる．この画面で，まず，コンパイルすべきブロックを設定し，それぞれの外部入力 (in1, in2) を反転表示させた状態で，外部入力を構成する信号線にチェックを入れ，それぞれの定数項，最小値，最大値を設定する．ここでは，仮数部の最小値を-128，最大値を 127 に，指数部を最小値 0，最大値 3 とし，定数部は双方ゼロとしている．また，w/err にチェックを入れ，誤差ありとしている．なお，数値属性は，ディレクティブで与えてもよい．

この操作を in1，in2 の双方に対しておこない，close ボタンを押すとコンパイルが開始され，図 5 に示すコンパイル結果が表示される．なお，入力属性は，コンパイラディレ

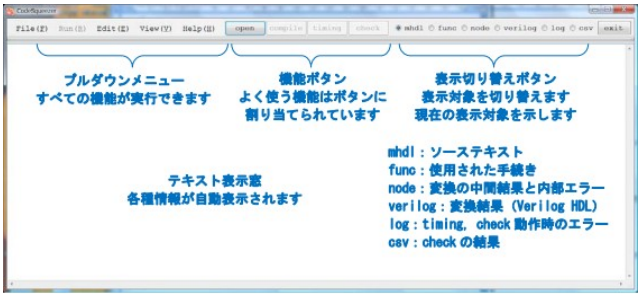


図 3　CodeSqueezer の起動画面

Figure 3.　Startup Screen of CodeSqueezer

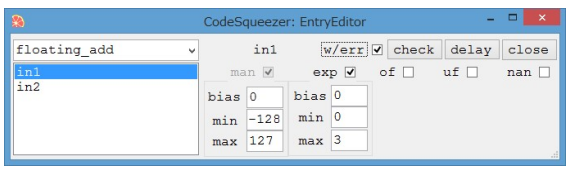


図 4　外部入力の型属性設定画面

Figure 4.　Type Setting Screen for External Input

クティブによって与えることもできる．

次いで，timing ボタンを押すと，タイミング制約の解決に必要なレジスタが挿入され，結果が表示される（図 6）．図 5 および図 6 の Verilog HDL ソースコード中には，コメントの形で，左辺信号線の整数値属性（最小値，最大値，定数値）が記述されている．末尾に文字“e”を付した値は誤差を含むことを意味する．

5. ストアードプログラム向けの FPGA

5.1 FPGA SRAM のアドレス空間へのアサイン

今日の FPGA は主に組込み分野での利用を念頭に設計されており，一般の科学技術計算への応用は容易ではない．FPGA でストアードプログラム方式の演算装置を構成するためには，動作中の論理書き換えが必要になる．

FPGA の論理を任意に書き換える簡単な手法は，FPGA の内部 SRAM をメモリバスに接続して，図 7 のように，アドレス空間にアサインする手法である．こうすれば，FPGA 上の任意の領域を CPU などから書き換え可能となる．FPGA の動作を規定する SRAM は，主記憶にロードされた CPU の実行コード領域に，FPGA 上のデータ格納用 RAM はデータ領域に相当しており，OS からは通常の CPU 実行コードに類似した扱いも可能と思われる[5].

5.2 オーバーレイ

今日のコンピュータでは，記憶デバイス上の小さな物理メモリ上に，巨大なアドレス空間を割り当てる仮想記憶技術が使用されている．FPGA の論理をメモリデバイスと同様に扱うことができるなら，仮想記憶に類似した技術により，FPGA の小規模な論理エレメントを用いて大規模な演算論理を仮想的に構築することも可能となる．

パイプライン演算装置は，データフローマシンとして動作し，入力側から出力側に向かって順次データが伝達され

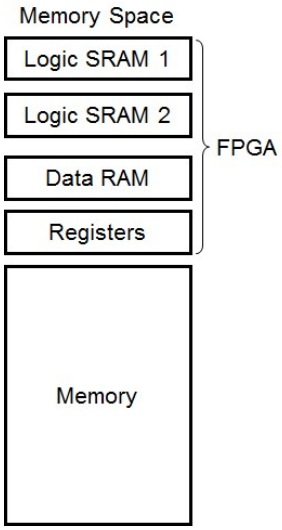
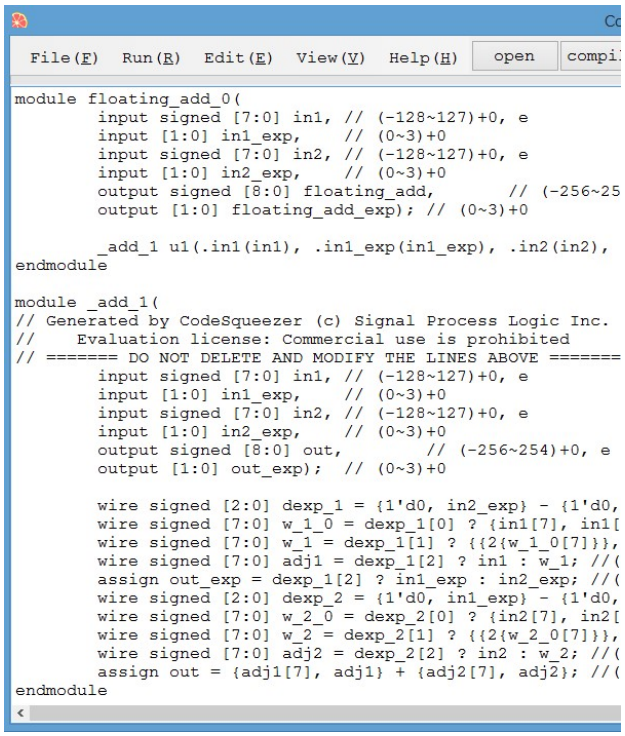


図 7　メモリ空間にアサインされた FPGA の RAM

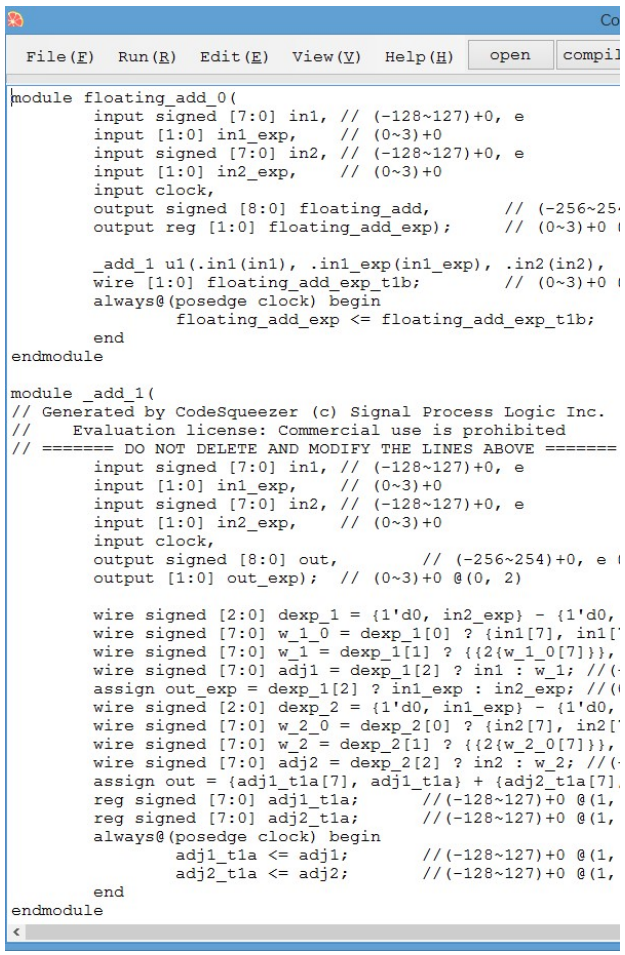
Figure 7　FPGA RAM Assigned to the Memory Space



```
File(F)  Run(R)  Edit(E)  View(V)  Help(H)  open  compi
module floating_add_0(
        input signed [7:0] in1, // (-128~127)+0, e
        input [1:0] in1_exp,    // (0~3)+0
        input signed [7:0] in2, // (-128~127)+0, e
        input [1:0] in2_exp,    // (0~3)+0
        output signed [8:0] floating_add,       // (-256~25
        output [1:0] floating_add_exp); // (0~3)+0

        _add_1 u1(.in1(in1), .in1_exp(in1_exp), .in2(in2),
endmodule

module _add_1(
// Generated by CodeSqueezer (c) Signal Process Logic Inc.
//    Evaluation license: Commercial use is prohibited
// ======= DO NOT DELETE AND MODIFY THE LINES ABOVE =======
        input signed [7:0] in1, // (-128~127)+0, e
        input [1:0] in1_exp,    // (0~3)+0
        input signed [7:0] in2, // (-128~127)+0, e
        input [1:0] in2_exp,    // (0~3)+0
        output signed [8:0] out,        // (-256~254)+0, e
        output [1:0] out_exp);  // (0~3)+0

        wire signed [2:0] dexp_1 = {1'd0, in2_exp} - {1'd0,
        wire signed [7:0] w_1_0 = dexp_1[0] ? {in1[7], in1[
        wire signed [7:0] w_1 = dexp_1[1] ? {{2{w_1_0[7]}},
        wire signed [7:0] adj1 = dexp_1[2] ? in1 : w_1; //(
        assign out_exp = dexp_1[2] ? in1_exp : in2_exp; //(
        wire signed [2:0] dexp_2 = {1'd0, in1_exp} - {1'd0,
        wire signed [7:0] w_2_0 = dexp_2[0] ? {in2[7], in2[
        wire signed [7:0] w_2 = dexp_2[1] ? {{2{w_2_0[7]}},
        wire signed [7:0] adj2 = dexp_2[2] ? in2 : w_2; //(
        assign out = {adj1[7], adj1} + {adj2[7], adj2}; //(
endmodule
```

図 5　コンパイル結果（部分）

Figure 5.　Compilation Results



```
File(F)  Run(R)  Edit(E)  View(V)  Help(H)  open  compil
module floating_add_0(
        input signed [7:0] in1, // (-128~127)+0, e
        input [1:0] in1_exp,    // (0~3)+0
        input signed [7:0] in2, // (-128~127)+0, e
        input [1:0] in2_exp,    // (0~3)+0
        input clock,
        output signed [8:0] floating_add,       // (-256~25
        output reg [1:0] floating_add_exp);     // (0~3)+0

        _add_1 u1(.in1(in1), .in1_exp(in1_exp), .in2(in2),
        wire [1:0] floating_add_exp_t1b;        // (0~3)+0
        always@(posedge clock) begin
                floating_add_exp <= floating_add_exp_t1b;
        end
endmodule

module _add_1(
// Generated by CodeSqueezer (c) Signal Process Logic Inc.
//    Evaluation license: Commercial use is prohibited
// ======= DO NOT DELETE AND MODIFY THE LINES ABOVE =======
        input signed [7:0] in1, // (-128~127)+0, e
        input [1:0] in1_exp,    // (0~3)+0
        input signed [7:0] in2, // (-128~127)+0, e
        input [1:0] in2_exp,    // (0~3)+0
        input clock,
        output signed [8:0] out,        // (-256~254)+0, e
        output [1:0] out_exp);  // (0~3)+0 @(0, 2)

        wire signed [2:0] dexp_1 = {1'd0, in2_exp} - {1'd0,
        wire signed [7:0] w_1_0 = dexp_1[0] ? {in1[7], in1[
        wire signed [7:0] w_1 = dexp_1[1] ? {{2{w_1_0[7]}},
        wire signed [7:0] adj1 = dexp_1[2] ? in1 : w_1; //(
        assign out_exp = dexp_1[2] ? in1_exp : in2_exp; //(
        wire signed [2:0] dexp_2 = {1'd0, in1_exp} - {1'd0,
        wire signed [7:0] w_2_0 = dexp_2[0] ? {in2[7], in2[
        wire signed [7:0] w_2 = dexp_2[1] ? {{2{w_2_0[7]}},
        wire signed [7:0] adj2 = dexp_2[2] ? in2 : w_2; //(
        assign out = {adj1_t1a[7], adj1_t1a} + {adj2_t1a[7],
        reg signed [7:0] adj1_t1a;      //(-128~127)+0 @(1,
        reg signed [7:0] adj2_t1a;      //(-128~127)+0 @(1,
        always@(posedge clock) begin
                adj1_t1a <= adj1;       //(-128~127)+0 @(1,
                adj2_t1a <= adj2;       //(-128~127)+0 @(1,
        end
endmodule
```

図 6　タイミング制約解決後（部分）

Figure 6.　After Timing Constraint Solved

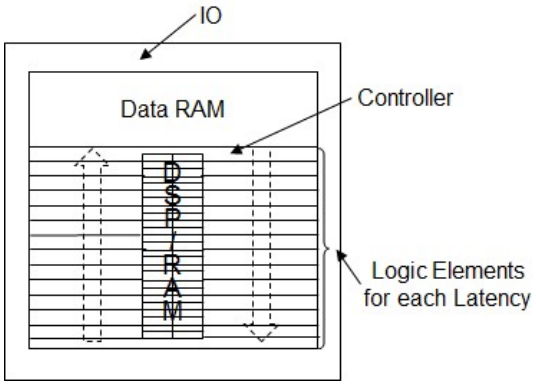


図 8　オーバーレイ用 FPGA

Figure 8　　FPGA for Overlay Technology

る．論理回路の間には適宜レジスタがおかれ，クロック信号に同期してデータが入力側から出力側へと伝達される．入力側の最上流から各論理回路までの間にデータが通過するレジスタの数をレイテンシと呼び，入力されたデータはレイテンシに相当するクロック数だけの遅れをもってそれぞれの論理回路に入力される．

演算の開始直後はレイテンシの小さな部分のみが動作し，演算の終了直前はレイテンシの大きな部分のみが動作するため，レイテンシに応じた演算論理の分割が合理的と考えられる．すなわち，レイテンシ m からレイテンシ m+n-1 までのデータを処理する n 段階の演算論理を FPGA にロードしてレイテンシ m のデータを処理し，演算の結果得られたレイテンシ m+n のデータを FPGA の内部 RAM に格納して次の段階の処理に備える構成がとりえよう．

図 7 では，論理を規定する領域を二つ設けている．こうすることにより，一方の領域が規定する論理で演算処理がなされている間に，他方の領域に次の段階で使用する論理を書き込むことができる．いずれの領域を論理規定に使用するかは，制御線により切り替える．さらに，図 8 に示したように，論理資源を複数の領域に分けてそれぞれ独立にこの制御をおこなえば，それぞれのブロックにレイテンシの異なる演算を割り当てることで，より無駄のない動作が可能となる．

このような技術は，CPU のメモリ管理技術との対比でみれば，仮想記憶よりもオーバーレイに近い技術である．しかしながら，CPU におけるオーバーレイはプログラマが明示的におこなうのに対し，パイプライン演算装置ではレイテンシという基準で機械的な処理が可能であり，利用者側からみれば，自動的に処理が行われる仮想記憶に近い使い勝手になると思われる．

6. まとめ

6.1 技術の利用分野

FPGA を用いたパイプライン数値演算装置は，必要な桁のみを処理するように演算器を構成することで，少ない論理資源を用いて高速に数値演算を実行することができる．本報では，この手法の実現方法とこれを用いたツールについて解説した．

FPGA は現在のところ組込み分野で主に用いられている．この場合，数値演算は FPGA の論理の一部として用いられる．このような用途には，Verilog HDL で記述されたモジュールの形での利用も現実的と考えている．

組込み分野向けの数値演算論理設計支援システムに関しては，現在評価版を公開しているが，誤差キャンセル対応機能が未実装で標準機能ブロックの提供も十分とはいえないなど，改良すべき点は多い．今後は，これらに対応した完全な形でのツールの提供を目指したい．

FPGA を用いたパイプライン方式の数値演算は，今日一般的に行われている CPU を用いた演算に比べて高いスループットが期待され，これを一般の科学技術計算に使用することができれば，その応用分野は大きく広がろう．

このような目的には，固定プログラミング方式での使用を前提とした今日の FPGA をそのまま用いることは難しいものと思われ，本報で提案した内部 RAM のメモリ空間へのマッピングなど，デバイス自体の変更も必要となろう．また，メモリのオーバーレイと同様な論理資源拡張手法も可能なデバイス構造とすれば，今日の技術レベルで製造可能な規模の FPGA も，大規模な数値演算処理用のデバイスとなりえるのではないかと期待している．

6.2 技術のトレンドとパイプライン処理

CPU が生まれた当時は，実現可能な論理規模の制約から，単一の万能演算器で複雑な演算を実行する CPU という技術は極めて有効であった．しかしながら今日では，大規模な論理回路も容易に入手可能となり，他の方式に基づく演算装置も実現可能となっている．

パイプライン処理は，今日，ストレージやデジタルテレビの分野で広く用いられている．これらに共通する要求は，高いスループットとローコストであり，これらは程度の差こそあれ，一般のコンピュータでも求められている．ストアードプログラム方式のコンピュータに利用可能な FPGA が登場すれば，パイプライン方式の経済的優位性が意味をもつものと期待している．

広く産業分野を見渡せば，化学産業における大量生産に伴うバッチプロセスから連続プロセスへの切り替え，組み立て産業におけるコンベア生産方式による生産性の向上など，パイプライン処理と同様の方式への切り替えによりスループットを向上させた例は多い．数値演算分野においても，CPU からパイプライン処理に切り替えることは，スループットの増大への有効な対応策といえよう．

6.3 CPU との相違点

FPGA を用いたパイプライン方式の数値演算装置は，CPU を用いた従来の方式とはかなり異なる特性をもつ．

最大の特徴は，多数の演算器を並列動作させることによ

る高速性であり，デバイスの集積度の上昇に応じた並列動作の規模拡大も容易である点も長所といえよう．

今回提案した手法では，数値表現に従来と異なる柔軟性の高い表現方法を用いている．これにより，人手による数値型の指定は不要となり，従来の CPU 向けのコンピュータ言語と比較してさらに抽象度の高い記述が可能となった．また，このような数値型を用いることで，有効桁のみを処理する論理回路の自動形成を可能としている．

従来の CPU では必然的に生じていた無効桁の演算が不要となることは，論理演算素子の有効活用だけでなく，電力消費の低減も期待される．

コンピュータが科学技術分野の数値計算に利用され始めた当初は，十進換算 7.22 桁の分解能をもつ単精度浮動小数点数が一般的に使用されていた．科学技術計算に使用される数値の有効桁数は十進で 3〜6 桁程度のものが大部分であり，これを表現することは，単精度浮動小数点数でも十分に可能である．しかしながら，コンピュータを利用する過程で桁落ちによると思われる問題が頻繁に発生したことから，今日では十進換算 15.95 桁の分解能をもつ倍精度浮動小数点数が広く利用されている．

この背景には，今日の科学技術計算に要求される精度が厳しくなったという事情もあろうが，まれに生じる桁落ちによる悪影響を回避するために膨大な演算が無駄に行われていることもまた事実であろう．

CPU と FPGA ではデバイス構造が異なるため単純に比較することは難しいが，必要な桁のみを処理するパイプライン演算装置に切り替えることでコンピュータの電力消費が大きく削減される可能性は高いものと思われる．

誤差キャンセルの問題は，倍精度浮動小数点表現を用いる CPU ではさほど問題となってはいない．これは，倍精度浮動小数点表現では仮数部の下位に余分なビットが含まれていることによると考えられる．しかしながら，倍精度浮動小数点表現が誤差キャンセルの問題に十分対応している保証はなく，この検証がなされていないまま使われていることは，今日の数値演算における問題点ともいえよう．

誤差キャンセルを検出するための個別誤差をトレースする演算は，FPGA 上の演算論理に含めておこなうことも可能である．このような演算論理は，演算結果として値と誤差の双方が得られ，演算結果の信頼性把握を可能とする．同様の個別誤差演算機能は，CPU にも組込むことができ，誤差情報を出力し誤差キャンセルの問題を警告する CPU も実現可能である．

計算結果に含まれる誤差を把握する手法は，それ自体で演算結果の信頼性を保証する意味でも有意義であろう．今日では，ローレンツ方程式に代表される，解に誤差を含みやすい方程式の存在が知られている[7]．また，静磁界のスカラポテンシャルによる解法などのように，一般的なポテンシャル問題の数値解法においても扱う数値の大きさが非常に異なるために桁落ちが発生しやすい分野もある[8]．演算結果に含まれる誤差が値と同時に得られれば，これらの難しい問題を扱う際にも，早期に問題点の所在把握が可能となり，数値計算を用いる研究開発も効率化されよう．

6.4 今後の課題

この研究は，2009 年の開始以来，機会をとらえて技術発表をおこない，実際に動作するツールをリリースしてきた．誤差のキャンセルに関しては，技術発表のみをおこない，これを実装したソフトウエアは現在開発段階にある．今後は，この機能を含む実用的なツールの開発に注力し，早期のリリースを実現したい．

現在公開しているツールは，組込み用途向けを想定したもので，FPGA に組込む数値演算モジュールを簡単な操作で作成するためのツールと位置付けている．

しかしながら，これに用いた数値演算論理形成手法は，組込み用途向けにとどまらず，一般的な科学技術計算向けの数値演算にも応用可能と考えている．このためには，FPGA 側にもいくつかの変更が必要である．この報告では，コンピュータシステム内部での利用に適した FPGA の構造に関して提案を行っているが，有限要素法をはじめとする幅広い科学技術計算に FPGA を利用する際には，その効率化のために RAM のマルチポート化などのいくつかの対応も求められるものと思われる．

CPU は長い歴史をもち，これに用いられる技術も高度に洗練されている．これに比べれば，FPGA を用いたパイプライン方式の数値演算装置に関しては，ソフトウエア，ハードウエアの双方に，まだまだ大きな課題が多数残されていよう．今後のパイプライン方式の数値演算装置の応用分野展開にあわせ，コンパイラやコード化技法に関しても，これに対応すべく技術開発をおこなっていきたい．

参考文献

[1] 瀬尾「浮動小数点処理を含む論理設計支援システム」DA シンポジウム 2010 論文集, p.3-8 (2010)

[2] 瀬尾「パイプライン処理のための演算仕様記述言語 mhdl とその処理系」DA シンポジウム 2012 論文集, p.115-120 (2012)

[3] 瀬尾「要因別誤差追跡に基づく安全な数値演算論理の自動設計」DA シンポジウム 2014 論文集, p.157-162 (2014)

[4] http://signal-process-logic.com/jp/CSEval/index.shtml

[5] 瀬尾「プログラマブルロジックデバイスおよびこれを用いたコンピュータ」特開 2015-091045 (2015)

[6] 瀬尾「誤差情報を含む浮動小数点表現とこれを用いた数値演算論理」第 76 回全国大会講演論文集，情報処理学会 (2014)

[7] E.N.Lorenz (杉山他訳)「ローレンツ／カオスのエッセンス」共立出版 (1997)

[8] J. Simkin & C. W. Trowbridge: "On the use of the total scalar potential in numerical solution of field probrems in electromagnetics", Int. J. Num. Meth. Eng., 14, 23 (1979)