浮動小数点処理を含む論理設計支援システム

シグナル・プロセス・ロジック株式会社

瀬尾雄三

要旨

浮動小数点数を用いれば、有効桁数に基づいた簡素なアルゴリズムによる数値型の自動決定が可能となる. これを応用して、抽象的な論理設計仕様からハードウエア記述を自動的に形成する手法を開発した. 浮動小数点数は固定小数点数と比較してダイナミックレンジが広いという特徴があるが、論理規模の増大という短所もある. この短所は仮数部と指数部のビット長を任意に設定可能とすることで軽減される. 本技術は、論理設計支援システムを目的として開発されたが、並列演算全般への応用も期待される. しかし、IEEE754 標準の浮動小数点演算とは異なって有効桁以下の誤差キャンセルが機能しないことから、その応用に際しては解法アルゴリズムの検討も必要と思われる.

Logic Design Support System Including Floating-Point Processing

Signal Process Logic Inc.

Yuzo Seo

Abstract

Using floating-point numbers, numeric types are automatically determined by simple algorithm based on the significant figure. Applying the algorithm, we developed an automatic conversion method from abstract logic design to the hardware description. Compared to fixed-point numbers, floating-point numbers have advantage of wide dynamic range but disadvantage of logic size expansion. The disadvantage is reduced by controlling bits length of both mantissa part and exponent part. The technology was developed for logic design support system but is also expected to be applied to general parallel computing. However, its lack of error cancellation at bits less than the significant figure, different from IEEE754 standard, will require reexamination of solution algorithms.

1. はじめに

浮動小数点表現はダイナミックレンジが広いという特徴があり、デジタル信号処理の分野でも利用が拡大している. 現在固定小数点演算が主流であるFPGAなどの並列演算においても、いずれ浮動小数点演算が一般的となる可能性は高い.

FPGA で浮動小数点数を扱う試みはいくつか報告されている. 最近のデバイスを用いればIEEE754 標準の浮動小数点演算論理を組み込むことも充分に可能である. しかし、論理資源の利用効率を考えると、指数部と仮数部のビット幅を任意に選択可能な自由度の高い浮動小数点表現形式が有利であり、数値型の複雑化に対応して最適な数値型を自動的に決定するシステムとの組合せも望まれている[1, 2, 3].

信号処理装置では、計測値など、有効桁数が限られた数値情報も多く扱われる. 有効な情報を含むビットのみを後段に伝達するとの判断基準を設ければ、演算器の入力数値型から出力数値型を自動的に決定することができる. この場合、乗算

結果の有効桁を取り出す処理に浮動小数点数を扱う必要が生じる.

数値型の自動決定は、抽象度の高い論理記述をハードウエア記述に変換する際にも必要となる. 単純な規則で数値型が自動決定される浮動小数点演算論理は、抽象的論理仕様記述と組み合わせることで相乗効果を発揮する.

このような考察に基づき、我々は抽象度の高い論理仕様をハードウエア記述に変換する論理設計支援システムへの、浮動小数点表現を用いた簡素な型決定アルゴリズムの応用を検討した. この論文では、そのための構造と変換アルゴリズムを中心に詳しく解説する.

2. 数値型

2.1. 数値型の表現方法

論理仕様の抽象記述は単純演算機能(**ファンクション**)を抽象的な信号線(**リンク**)で接続した**ファンクション・ダイアグラム**の形式で表現される.

任意の数値型で表現された信号を伝達するリンクは、class Link として以下のリストに示す形に定義され、Verilog HDL の wire に相当する信号線(**ワイヤ**:class Wire)と関連付けられる[1].

```
bool certain;  // 誤差なしフラグ
Wire^ man;  // 仮数部
Wire^ exp;  // 指数部
Wire^ of;  // オーバーフローフラグ
Wire^ uf;  // アンダーフローフラグ
Func^ src; // 信号源へのポインタ
List<Func^>^ dests; // 信号の行き先
```

誤差なしフラグはこのリンクで伝達される数値に含まれる誤差の有無を示すもので、数値型を自動決定する際に使用される. リンクの値は仮数部と指数部を表す二つのワイヤ(man および exp)で表現する. ワイヤは整数値を表し、リンクの値は $man \times 2^{exp}$ となる.

異常状態は二つのワイヤ of および uf により表示される. of のみが 1 の場合はオーバーフロー、uf のみが 1 の場合はアンダーフロー、双方が 1 の場合は NaN(数値化不能)であることを示す.

IEEE754 標準の浮動小数点数は+0 と–0 に異なる符号を割り当てているが、負数を 2 の補数表現する本仕様では 0 は符号をもたない. 0 の逆数は NaN と定義され、±0 の逆数を±∞とする IEEE754 標準とは異なる結果を与える. この違いによって問題が生じることはないと考えているが、同等性を要求する際には注意が必要である.

ワイヤの属性を表す class Wire は以下のように定義されている.

```
String^ name;  // ワイヤ名
bool neg;  // 符号反転フラグ
long long bias, min, max;
Node^ src; // 信号源
List<ModWire^>^ dests; // 信号の行き先
WirePair^ inport; // 信号源となる入力ポート
WirePair^ outport; // 出力ポートに接続
int clock_delay; // クロック遅延累積値
double logic_delay; // 論理遅延累積値
```

符号反転フラグは、信号線のビットパターンが符号を反転してコード化されていることを示す. 定数 bias と信号線の値の和がワイヤの値となる. min および max がともに 0 の場合、ワイヤは定数を表し、デバイス上の信号線は形成されない.

符号の有無と信号線幅は、信号線の値の最小値(min)と最大値(max)から一意に決定される. 整数の型は符号の有無とビット長で規定するのが一般的だが、値の範囲を型属性とするこの方法は、演算結果の型決定が容易であり、ビット長の異なる数値間の加減算が多い場合には信号線本数も削減されるという特徴がある.

class Wire には、接続情報を格納するためのポインタと信号遅延を格納する変数が定義されている. これらに関しては後述する.

2.2. 数値型の自動決定

演算に用いる信号のいずれかが誤差を含む場合には、演算結果の有効な桁のみを後段に伝達する. このような手法は**有効桁数**という概念で一般に説明されるが、あいまいさを排除するため、

[1]以下、ソースリストは Microsoft 社の Visual C++および.NET Framework[4]を使用して記述する

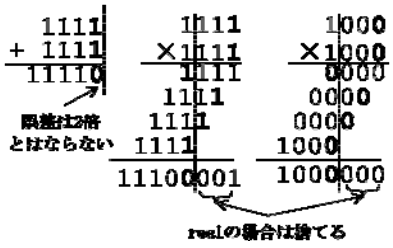


図 1 不確定性指数が管理範囲となる桁

“桁の重みに対する誤差の比率”で定義される**不確定性指数**が一定範囲内の桁を最下位とするように下位ビットを切り捨てることとした. 不確定性指数の管理範囲は保護ビットにも配慮して定める.

図 1 は加算と二種類の乗算について、不確定性指数が管理範囲に含まれる桁を太字で示している. 二つの誤差の和は 2 倍とはならず、加算の際には最下位が残る. 乗算の際には、無効な桁数は一定で、有効な桁数が変化する. 各桁の不確定性指数を評価することで、切り捨てるべき桁数が合理的かつ機械的に判定される.

各ファンクションのコード化に際しては、有効な情報が失われることのないよう、安全な側で下位ビットを切り捨てるため、演算を繰り返すに従って最下位桁に含まれる誤差が増加する. これを補正するため、必要に応じて有効桁数を切り捨てるファンクションを挿入する.

3. データ構造

3.1. ファンクション・ダイアグラム

論理仕様はファンクションをリンクで接続したファンクション・ダイアグラムの形で抽象的に記述することができる. ファンクション・ダイアグラムの要素を図 2 に示す. ファンクションのデータ構造(class Func)は以下のように定義されている.

```
String^ name; // ファンクションの名称
List<Port^>^ inports; // 入力ポート
List<Port^>^ outports; // 出力ポート
List<Func^>^ funcs; // 下位定義*
Coder coder; // コード化関数
```

ポート(class Port)は以下の要素からなる.

```
String^ name; // ポート名
Link^ link; // 接続された信号線
Func^ func; // 下位定義*
```

論理仕様定義もファンクションとなる. この場合、定義に含まれるファンクションを上記リストで*を付した変数(funcs とポートの func)に記述する.

coder はファンクションのもつ機能をコード化するための関数で、各ファンクションに固有の関数が与えられている. これについては後述する.

3.2. ノードとノードグラフ

ノードは、パイプライン化の最小単位となる演算要素であり、FPGA の論理要素 1 段に対応するように形成する. ノードをワイヤで接続してノードグラフを形成する. ノードグラフの要素を図 3 に示す.

class Node の定義を以下に示す.

```
Wire^ left_wire; // 左辺のワイヤ
String^ code_str; // 式の文字列
List<ModWire^>^ right_wires; // 右辺
int clock_delay; // クロック遅延
double logic_delay; // 論理遅延時間
```

ノードは Verilog HDL の代入文に対応しており、left_wire がその左辺に相当する. 右辺の式は、フォーマット文字列(標準 C の場合は"%s + %s”など)を code_str に、式中のワイヤを修飾ワイヤリスト right_wires に記述する形で表現する.

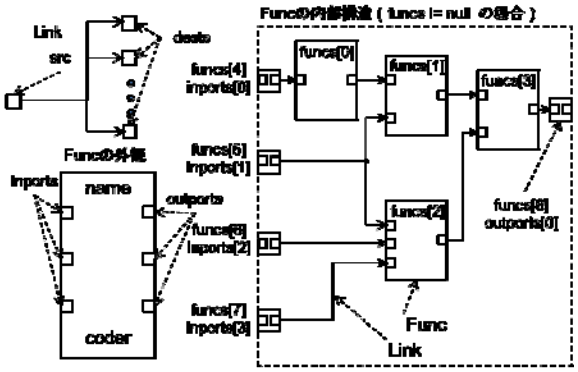


図 2 ファンクション・ダイアグラムの要素

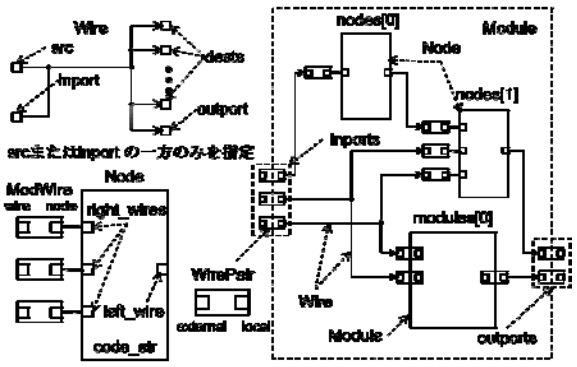


図 3 ノードグラフの要素

修飾ワイヤ(class ModWire)は、ワイヤを部分ビット列の形あるいは桁を拡張した形で参照するためのもので、次の要素を含んでいる.

```
Wire^ wire; // ワイヤ
String^ ope; // リダクション演算子
int ex_left, ex_right; // ビット拡張幅
Node^ node; // これを使用しているノード
```

ex_left, ex_right は左右へのビット拡張幅であり、これらの数値が正の場合は桁拡張を、負の場合はビットの切り捨てを行う. ope はリダクション演算子で、ビット幅が1を超える場合に付加される. class ModWire には、ワイヤ名にこれらの修飾を加えて Verilog HDL の書式に従う文字列に変換する、文字化関数が定義されている.

Verilog HDL はモジュール単位でソースコードを記述する. これに対応して、ノードグラフもモジュール単位に分割して形成する. class Module の要素は以下のとおりである.

```
String^ name; // モジュール名
List<WirePair^>^ inports; // 入力ポート
List<WirePair^>^ outports; // 出力ポート
List<Node^>^ nodes; // ノードのリスト
List<Module^>^ modules; // サブモジュール
```

class WirePairは二つのワイヤの対であり、モジュールの入出力信号線とモジュールを配置した際の上位ワイヤ接続を対とすることで、モジュール内外の信号線接続関係が規定される.

4. 変換アルゴリズム

4.1. ファンクションのコード化

class Func にはこれをハードウエア記述に変換するための関数が付随している. Coder はその関数型であり、以下の形で定義されている.

```
typedef bool (*Coder)(
    Module^ module, Func^ func);
```

コード化関数は、全ての入力信号線の型が決定している場合に、出力信号線の型を決め、ファンクションをノードグラフに変換して引数 module の要素 nodes に格納して、true を返す.

図 4 に二項加算を行うファンクションのコード化関数の処理内容を示す. 入力信号の誤差の有無(int: 誤差なし整数、real: 誤差あり実数)の組合せ毎に、ワイヤのレベルでの信号の処理フローが規定されている.

ファンクション・ダイアグラムのコード化は、信号の流れに即してワイヤレベルのコード化関数(矩形で表示)を順次呼び出すことで遂行される. ワイヤレベルのコード化関数は、入出力ワイヤを引数に与えて呼び出されると、出力ワイヤの型を設定し、所定の演算を行うノードを形成する.

ワイヤレベルのコード化関数の一例として、ワイヤの加算を行う関数の処理内容を図 5 に示す. ワイヤが定数であるか変数であるかを判定して適切なノードを形成する. この過程で出力ワイヤの属性値が決定され、その組合せであるファンクション出力リンクの数値型も自動的に決定される.

4.2. タイミング処理

タイミングはノードグラフの段階で解決される. class Node にはノードを経由することによる信号の遅延がノード形成時に設定されている. class

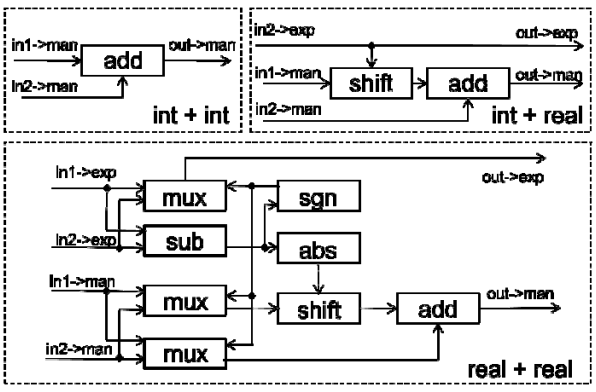


図 4 加算ファンクションのコード化処理

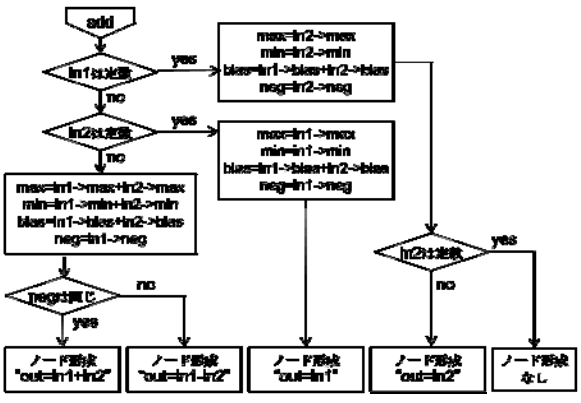


図 5 ワイヤレベルのコード化(add)

Wire には、信号が確定するタイミングを示す logic_delay（論理遅延）と clock_delay（クロック遅延）情報が与えられる.

ノードグラフの信号を入力側からたどり、ノードを経由する毎に logic_delay にノードの論理遅延時間を加算する. ワイヤの入力端にレジスタが挿入されている場合には、論理遅延を 0 にリセットすると同時にクロック遅延を一つ増す. この操作を信号の伝達経路に沿って順次行うことで、各ワイヤの遅延状態が得られる.

この処理の過程で、第一に、ノードの全ての入力ワイヤのクロック遅延を一致させる. この操作は、入力ワイヤのクロック遅延を比較して、これが異なる場合には、遅延の小さいワイヤの入力端にレジスタを挿入することで行われる.

第二に、クロック周期に由来するタイミング制約を満足させる. この操作は、ノード出力部の論理遅延時間がタイミング制約を満足しない場合に、入力側に接続された全てのワイヤの入力端にレジスタを挿入することで行われる.

レジスタを挿入すべきワイヤが既に他のノードのタイミングチェックに使用されていた場合には、ワイヤの遅延情報を変更できないため、その出力側に代入演算を行う新たなノードを追加し、追加されたノードの出力にレジスタを挿入する.

4.3. Verilog HDL への変換

ノードグラフは機械的な手続きにより Verilog HDL のソースコードに変換される. モジュールのヘッダ部は、モジュール名と入出力ポートの local ワイヤ情報から作成される. モジュールに含まれるノードは nodes に格納されており、これに含まれるノードを順に出力する. それぞれのノードの Verilog HDL 記述は、ノードの左辺ワイヤが出力ポートに接続されているか否かと、レジスタが挿入されているか否かに応じて、図 6 に示す各種の形式に変換される.

outport	registered	Verilog HDL codes
nullptr	false	wire [msb:0] name = 式
nullptr	true	reg [msb:0] name name <= 式
defined	false	output [msb:0] name assign name = 式
defined	true	output reg [msb:0] name name <= 式

図 6 ノードの Verilog HDL への変換

モジュール内部に配置されるサブモジュール定義は modules に格納されている. サブモジュールが定義されている場合は、名前と入出力ポートリストからモジュール配置コードを形成して上位モジュール内に記述し、上位モジュール記述の後にサブモジュールの定義内容を記述する.

なお、Verilog HDL の言語仕様上は同じモジュール定義を複数個所に配置可能だが、今回構築したシステムではモジュール内部でのレジスタ挿入位置が配置毎に異なる可能性もあるため、配置ごとに個別にモジュール定義を与えている.

5. まとめと今後の課題

5.1. このシステムの位置付け

今回紹介したシステムは、浮動小数点数を直接扱うことが可能であり、ダイナミックレンジが広く安全性に優れた信号処理論理が形成できるという特徴がある. また、簡素なアルゴリズムで抽象的な論理記述からハードウエア記述に変換可能であることも、本システムの長所である.

浮動小数点演算で問題となる論理規模の増加は、高自由度数値型により抑制されている. しかし、固定小数点演算に比べれば論理規模は拡大し、遅れ時間も増す. 速度と効率が要求される分野では、固定小数点演算が有利であると思われる.

今回提案した方式は、アルゴリズムが簡素で、簡易で安価なシステムを構築できる. 当面は、この特徴を生かした論理設計支援システムとして製品化したいと考えている.

半導体技術の進歩とともに、いずれは並列演算分野での浮動小数点の利用も進むと思われる. 今後の本技術の利用分野拡大に期待したい.

5.2. 数値計算に応用する際の問題点

浮動小数点数を直接処理することができる論理開発環境は、FPGA を用いた数値演算アクセラレータなどの、並列演算プログラミング環境としても利用できよう.

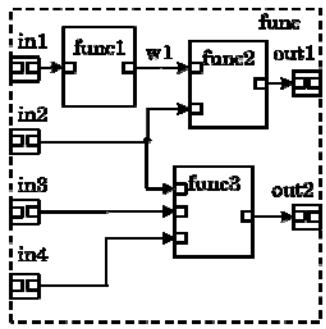
今回提案した浮動小数点表現形式は、無効な桁も扱う IEEE 754 標準の浮動小数点表現に比べて、論理規模が圧縮できるという利点がある. しかし無効な桁も誤差のキャンセルには有効に機能しており、これを切り捨てることで問題が生じる可能性がある.

前段で混入した誤差が後段で打ち消される演算処理がなされている場合、誤差情報を含む全ての演算結果を後段に伝達した場合には後段で誤差がキャンセルされるが、有効な桁のみを後段に伝達する方式では誤差の情報が失われ、後段での誤差のキャンセルが機能しない.

誤差のキャンセルが働かないという現象は、固定小数点に変換して演算した場合にも起こり、並列演算に共通する課題といえよう. この問題は、演算順序の最適化などの数値演算アルゴリズムの改良により解決可能と思われ、並列演算システムにおける一つの検討課題であると考えている.

5.3. 抽象的論理仕様記述言語

この論文では、抽象的論理仕様をハードウエア記述に変換するためのデータ構造とアルゴリズムを中心に述べたが、信号処理論理設計支援システムには、抽象的論理仕様を効率的に入力する手段も必要になる. 最も効率的な仕様記述方法は言語形式であると考えられることから、このための言語仕様についても検討している.




```
func.(out1, out2) (in1, in2, in3, in4)
        w1 = func1(in1)
        out1 = func2(w1, in2)
        out2 = func3(in2, in3, in4)
```

図 7 MHDL による論理仕様記述

この言語は、図 7 に示すように、ファンクション名と入出力ポートを定義する**ファンクション宣言文**と、ファンクションとリンクと関連付ける**代入文**とから構成される. “代入文の右辺に使用するリンクは、入力ポートとして定義されているか、それ以前の行の代入文左辺に現れていること”という制約を設け、信号の流れを記述順序と一致させる.

本言語の詳細は検討中であるが、右辺への数式記述やレジスタファイル、ステートマシンの取り扱いなどを含めて規格化したいと考えている.

本言語規格の制定・管理のための非営利団体設立を想定して URL “http://mhdl.org”を確保している. MHDL はこの言語の仮の名称であり、“Minimized (または META) Hardware Description Language”の意味合いを込めた. この抽象的論理仕様記述言語に関する情報は、今後、同ウエブサイトに掲載する予定である.

引用文献

[1] Dido, J., et.al. “A flexible floating-point format for optimizing data-paths and operators in FPGA based DSPs”, *Proceedings of the 2002 ACM/ SIGDA tenth international symposium on Field programmable gate arrays* (2002)

[2] Belanovic, P., “Library of Parameterized Hardware Modules for Floating-Point Arithmetic with An Example Application”, *http://www.ece.neu.edu/groups/rcl/theses/belanovic2002.pdf* (2002)

[3] Beauchamp, M.J., et.al, “Embedded Floating-Point Units in FPGAs”, *Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays* (2006)

[4] 矢吹太朗「VisualC++入門」日経 BP ソフトプレス(2009)